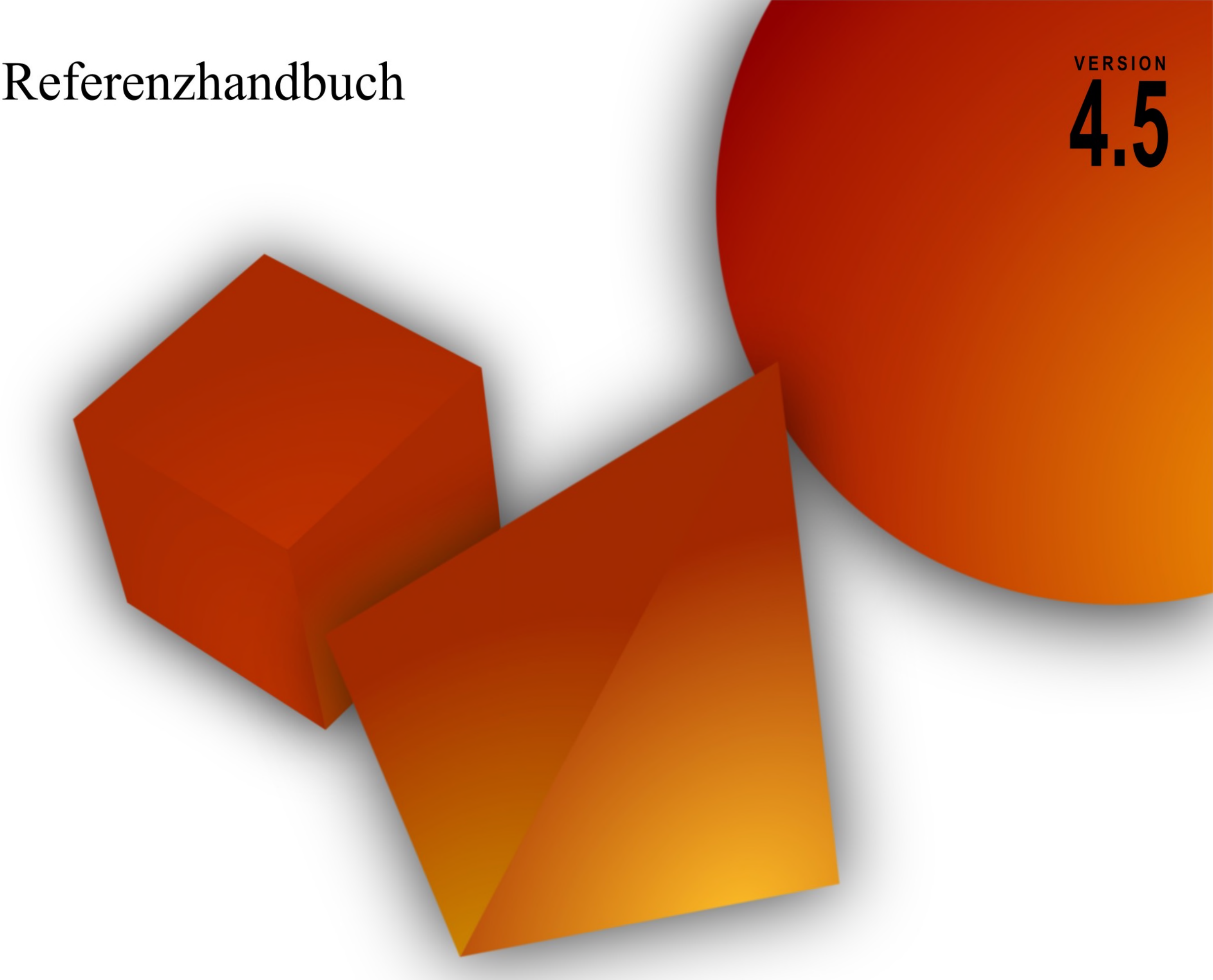
Referenzhandbuch

VERSION 4.5



Borland® C++

Borland C++ 4.5

Referenzhandbuch

BORLAND

Copyright © 1987, 1993 by Borland International, Inc.
Alle Rechte vorbehalten.

Alle Borland-Produkte sind Warenzeichen oder eingetragene Warenzeichen von Borland International, Inc. Andere aufgeführte Waren- und Produktbezeichnungen sind Warenzeichen oder eingetragene Warenzeichen der jeweiligen Rechteinhaber.

Einschränkung der Gewährleistung:
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber:

Borland GmbH
Monzastraße 4c
63225 Langen

Printed in Ireland.

Inhalt

Einführung . **19**

Kapitel 1 Bibliotheksreferenz **21**

Warum der Quelltext der Laufzeitbibliothek wichtig ist . 22
Die Laufzeitbibliotheken 22
 Die statischen Bibliotheken 23
 Die dynamischen Linkbibliotheken (DLL) . 25
Die Borland C++ Header-Dateien 26
Bibliotheksroutinen nach Kategorien geordnet . 28
 Routinen mit C++ Prototypen 28
 Klassifizierungs- Routinen . 28
 Routinen zur Ein-/ Ausgabe in Textbild- schirm- anwendungen . 29
 Konvertierungs- Rroutinen . 29
 Diagnoseroutinen 29
 Routinen zur Ver- zeichnisverwaltung . . 29
 EasyWin-Routinen 30
 Inline-Routinen 30
 Ein-/Ausgabe-Routinen 30
 Schnittstellenroutinen 32
 Von der locale-Einstellung betroffene API-Routinen . 32
 Manipulations- routinen 33
 Mathematische Routinen 33
 Speicherroutinen 34
 Verschiedene Routinen 34
 Veraltete Definitionen 35
 Routinen zur Prozeßsteuerung 35
 Zeit- und Datumsroutinen 36
 variable Argumentlisten *36*

Kapitel 2 Die Funktion main **37**

Die Parameter von main 37
 Die Argumente von main auswerten . . . 38
 Jokerzeichen in Kommandozeilen- parametern . 39
-p verwenden (Pascal-Aufrufkonventionen) 40
Der Rückgabewert von main 40
Übergabe von Datei-Informationen an Kindprozesse . 41
Programme mit mehreren Threads 41

Kapitel 3 Die Bibliotheksfunktionen . **43**

Beispielfunktion . 43
abort . 45
abs . 45
access . 46
acos, acosl . 46
alloca . 47
asctime . 48
asin, asinl . 48
assert . 49
atan, atanl . 49
atan2, atan2l . 50
atexit . 51
atof, _atold . 51
atoi . 52
atol . 53
_atold . 53
bdos . 53
bdosptr . 54
_beginthread . 55
_beginthreadNT . 56
biosequip . 57
_bios_equiplist . 58
bioskey . 59
biosmemory . 60
_bios_memsize . 60

biostime 61
_bios_timeofday 61
bsearch 62
cabs, cabsl 63
calloc 64
ceil, ceill 64
_c_exit 65
_cexit 65
cgets 66
_chain_intr 66
chdir 67
_chdrive 68
_chmod 68
chmod 68
chsize 69
_clear87 69
clearerr 70
clock 70
_close 71
close 71
closedir 71
clreol 72
clrscr 72
_control87 73
cos, cosl 74
cosh, coshl 74
country 75
cprintf 76
cputs 77
_creat 77
creat 77
creatnew 78
creattemp 79
_crotl, _crotr 80
cscanf 80
ctime 81
ctrlbrk 82
cwait 82
delline 84
difftime 84
disable, _disable, enable, _enable 85
div 85
_dos_close 86
_dos_commit 86
_dos_creat 86
_dos_creatnew 87
dosexterr 88
_dos_findfirst 89
_dos_findnext 90
_dos_getdate, _dos_setdate, getdate, setdate 91
_dos_getdiskfree 92
_dos_getdrive, _dos_setdrive 92
_dos_getfileattr, _dos_setfileattr 93
_dos_getftime, _dos_setftime 93
_dos_gettime, _dos_settime 94
_dos_getvect 95
_dos_open 95
_dos_read 97
_dos_setdate 98
_dos_setdrive 98
_dos_setfileattr 98
_dos_setftime 98
_dos_settime 98
_dos_setvect 98
dostounix 99
_dos_write 99
dup 100
dup2 101
ecvt 101
__emit__ 102
enable, _enable 103
_endthread 103
eof 104
exec... 104
_exit 107
exit 107
exp, expl 108
_expand 109
fabs, fabsl 109
farcalloc 109
farfree 110
farmalloc 111
farrealloc 111
fclose 112
fcloseall 112
fcvt 113
fdopen 113
feof 114
ferror 115
fflush 115

fgetc . 116
fgetchar . 116
fgetpos . 116
fgets . 117
filelength . 117
fileno . 118
findfirst . 118
findnext . 120
floor, floorl . 121
flushall . 121
_fmemccpy . 121
_fmemchr . 122
_fmemcmp . 122
_fmemcpy . 122
_fmemicmp . 122
_fmemmove . 122
_fmemset . 122
fmod, fmodl . 122
_fmovmem . 123
fnmerge . 123
fnsplit . 124
fopen . 125
FP_OFF, FP_SEG 126
_fpreset . 126
fprintf . 127
fputc . 128
fputchar . 128
fputs . 128
fread . 129
free . 129
freopen . 130
frexp, frexpl . 131
fscanf . 131
fseek . 132
fsetpos . 133
_fsopen . 133
fstat, stat . 135
_fstr* . 136
ftell . 137
ftime . 137
_fullpath . 138
fwrite . 139
gcvt . 139
geninterrupt . 139
getc . 140
getcbrk . 140
getch . 141
getchar . 141
getche . 142
getcurdir . 142
getcwd . 142
getdate . 143
_getdcwd . 143
getdfree . 144
getdisk, setdisk 145
getdta . 145
getenv . 146
getfat . 146
getfatd . 147
getftime, setftime 147
getpass . 148
_get_osfhandle 149
getpid . 149
getpsp . 149
gets . 150
gettext . 150
gettextinfo . 151
gettime, settime 152
getvect, setvect 152
getverify . 153
getw . 154
gmtime . 154
gotoxy . 155
_heapadd . 156
heapcheck . 156
heapcheckfree . 156
heapchecknode 157
_heapchk . 157
heapfillfree . 158
_heapmin . 158
_heapset . 159
heapwalk . 159
_heapwalk . 160
highvideo . 160
hypot, hypotl . 160
_InitEasyWin . 161
inp . 161
inport . 162
inportb . 162
inpw . 162

insline . 163
int86 . 163
int86x . 164
intdos . 165
intdosx . 165
intr . 166
ioctl . 167
isalnum . 169
isalpha . 169
isascii . 169
isatty . 170
iscntrl . 170
isdigit . 171
isgraph . 171
islower . 171
isprint . 172
ispunct . 172
isspace . 173
isupper . 173
isxdigit . 174
itoa . 174
kbhit . 175
labs . 175
ldexp, ldexpl . 175
ldiv . 176
lfind . 176
localeconv . 177
localtime . 179
lock . 180
locking . 181
log, logl . 182
log10, log10l . 183
longjmp . 183
lowvideo . 184
_lrotl, lrotr . 185
lsearch . 185
lseek . 186
ltoa . 187
_makepath . 187
malloc . 188
_matherr, _matherrl 189
max . 191
mblen . 192
mbstowcs . 192
mbtowc . 193
memccpy, _fmemccpy 194
memchr, _fmemchr 194
memcmp, _fmemcmp 195
memcpy, _fmemcpy 195
memicmp, _fmemicmp 196
memmove, _fmemmove 196
memset, _fmemset 197
min . 197
mkdir . 198
MK_FP . 198
mktemp . 199
mktime . 199
modf, modfl . 200
movedata . 200
movetext . 201
movmem, _fmovmem 201
_msize . 202
normvideo . 202
offsetof . 202
_open . 203
open . 203
opendir . 205
_open_osfhandle 206
outp . 206
outport, outportb 207
outpw . 207
parsfnm . 208
_pclose . 208
peek . 209
peekb . 209
perror . 209
_pipe . 211
poke . 212
pokeb . 212
poly, polyl . 213
_popen . 213
pow, powl . 214
pow10, pow10l . 215
printf . 216
putc . 222
putch . 222
putchar . 223
putenv . 223
puts . 224
puttext . 224

putw . . . 225
qsort . . . 226
raise . . . 227
rand . . . 227
random . . . 228
randomize . . . 228
_read . . . 229
read . . . 229
readdir . . . 230
realloc . . . 231
remove . . . 231
rename . . . 232
rewind . . . 232
rewinddir . . . 233
rmdir . . . 233
rmtmp . . . 234
_rotl, _rotr . . . 234
_rtl_chmod . . . 234
_rtl_close, close . . . 235
_rtl_creat . . . 236
_rtl_heapwalk . . . 236
_rtl_open . . . 237
_rtl_read . . . 239
_rtl_write . . . 239
scanf . . . 240
_searchenv . . . 247
searchpath . . . 248
_searchstr . . . 248
segread . . . 249
setbuf . . . 249
setcbrk . . . 250
_setcursortype . . . 251
setdate . . . 251
setdisk . . . 251
setdta . . . 251
setftime . . . 252
setjmp . . . 252
setlocale . . . 253
setmem . . . 257
setmode . . . 257
settime . . . 258
setvbuf . . . 258
setvect . . . 259
setverify . . . 259
signal . . . 260
sin, sinl . . . 263
sinh, sinhl . . . 264
sleep . . . 264
sopen . . . 265
spawn... . . . 266
_splitpath . . . 270
sprintf . . . 271
sqrt, sqrtl . . . 271
srand . . . 272
sscanf . . . 272
stackavail . . . 273
stat . . . 273
_status87 . . . 273
stime . . . 274
stpcpy . . . 274
strcat, _fstrcat . . . 275
strchr, _fstrchr . . . 275
strcmp, _fstrcmp . . . 276
strcmpi . . . 276
strcoll . . . 277
strcpy, _fstrcpy . . . 277
strcspn, _fstrcspn . . . 278
_strdate . . . 278
strdup, _fstrdup . . . 278
_strerror . . . 279
strerror . . . 280
strftime . . . 280
stricmp, _fstricmp . . . 282
strlen, _fstrlen . . . 283
strlwr, _fstrlwr . . . 283
strncat, _fstrncat . . . 284
strncmp, _fstrncmp . . . 284
strncmpi . . . 285
strncpy, _fstrncpy . . . 285
strnicmp, _fstrnicmp . . . 286
strnset, _fstrnset . . . 286
strpbrk, _fstrpbrk . . . 287
strrchr, _fstrrchr . . . 287
strrev, _fstrrev . . . 288
strset, _fstrset . . . 288
strspn, _fstrspn . . . 289
strstr, _fstrstr . . . 289
_strtime . . . 289
strtod, _strtold . . . 290
strtok, _fstrok . . . 291

strtol ... 292
_strtold ... 293
strtoul ... 293
strupr, _fstrupr ... 294
strxfrm ... 294
swab ... 295
system ... 296
tan, tanl ... 296
tanh, tanhl ... 297
tell ... 297
tempnam ... 298
textattr ... 299
textbackground ... 300
textcolor ... 301
textmode ... 302
time ... 303
tmpfile ... 304
tmpnam ... 304
toascii ... 305
_tolower ... 305
tolower ... 306
_toupper ... 306
toupper ... 307
tzset ... 307
ultoa ... 308
umask ... 309
ungetc ... 309
ungetch ... 310
unixtodos ... 310
unlink ... 311
unlock ... 311
utime ... 312
va_arg, va_end, va_start ... 313
vfprintf ... 314
vfscanf ... 314
vprintf ... 315
vscanf ... 316
vsprintf ... 317
vsscanf ... 317
wait ... 318
wcstombs ... 319
wctomb ... 320
wherex ... 320
wherey ... 321
window ... 321
_write ... 322
write ... 322

Kapitel 4 Globale Variablen ... 325

_8087 ... 325
_argc ... 326
_argv ... 326
_ctype ... 326
_daylight ... 327
_directvideo ... 327
_environ ... 327
errno, _doserrno, _sys_errlist, _sys_nerr ... 328
_floatconvert ... 330
_fmode ... 331
_new_handler ... 331
_osmajor, _osminor, _osversion ... 332
_psp ... 332
_threadid ... 333
__throwExeceptionName, __throwFileName, __throwLineNumber ... 333
_timezone ... 334
tzname ... 334
_version ... 334
_wscroll ... 335

Kapitel 5 Die C++ Klassen für Ein-/Ausgabe-Streams ... 337

Klasse conbuf ... 337
 Public-Konstruktoren ... 337
 Public-Elementfunktionen ... 338
Klasse constream ... 339
 Public-Konstruktoren ... 339
 Public-Elementfunktionen ... 339
Klasse filebuf ... 339
 Public-Konstruktoren ... 340
 Public-Datenelemente ... 340
 Public-Elementfunktionen ... 340
Klasse fstream ... 342
 Public-Konstruktoren ... 342
 Public-Elementfunktionen ... 343
Klasse fstreambase ... 343
 Public-Konstruktoren ... 343
 Public-Elementfunktionen ... 343
Klasse ifstream ... 344

Public-Konstruktoren 344
Public-Elementfunktionen 345
Klasse ios . 345
Public-Datenelemente 345
Protected-Datenelemente 346
Public-Konstruktoren 346
Protected-Konstruktoren 346
Public-Elementfunktionen 347
Protected-Elementfunktionen 349
Klasse iostream . 349
Public-Konstruktoren 349
Klasse iostream_withassign 349
Public-Konstruktoren 349
Public-Elementfunktionen 349
Klasse istream . 350
Public-Konstruktoren 350
Public-Elementfunktionen 350
Protected-Elementfunktionen 352
Klasse istream_withassign 352
Public-Konstruktoren 352
Public-Elementfunktionen 352
Klasse istrstream . 352
Public-Konstruktoren 352
Klasse ofstream . 353
Public-Konstruktoren 353
Public-Elementfunktionen 353
Klasse ostream . 354
Public-Konstruktoren 354
Public-Elementfunktionen 354
Klasse ostream_withassign 355
Public-Konstruktoren 355
Public-Elementfunktionen 355
Klasse ostrstream . 355
Public-Konstruktoren 355
Public-Elementfunktionen 356
Klasse streambuf . 356
Public-Konstruktoren 356
Public-Elementfunktionen 356
Protected-Elementfunktionen 357
Klasse strstreambase 359
Public-Konstruktoren 359
Public-Elementfunktionen 359
Klasse strstreambuf 359
Public-Konstruktoren 359
Public-Elementfunktionen 360
Klasse strstream . 361
Public-Konstruktoren 361
Public-Elementfunktionen 361

Kapitel 6 Klassen und Makros für persistente Streams 363

Klasse fpbase . 364
Public-Konstruktoren 364
Public-Elementfunktionen 364
Klasse ifpstream . 364
Public-Konstruktoren 365
Public-Elementfunktionen 365
Klasse ipstream . 365
Public-Konstruktoren 365
Public-Elementfunktionen 366
Protected-Konstruktoren 367
Protected-Elementfunktionen 367
Friend-Elemente 368
Klasse ofpstream . 368
Public-Konstruktoren 368
Public-Elementfunktionen 369
Klasse opstream . 369
Public-Konstruktoren 369
Destruktor . 369
Public-Elementfunktionen 369
Protected-Konstruktoren 371
Protected-Elementfunktionen 371
Friend-Elemente 372
Klasse pstream . 372
Typdefinitionen 372
Public-Konstruktoren 372
Destruktor . 372
Public-Elementfunktionen 373
Protected-Datenelemente 374
Protected-Konstruktoren 374
Protected-Elementfunktionen 374
Klasse TStreamableBase 374
Typdefinitionen 375
Destruktor . 375
Public-Elementfunktionen 375
Klasse TStreamableClass 375
Public-Konstruktoren 375
Friend-Klassen . 376
Klasse TStreamer . 376

Public-Elementfunktionen 376
Protected-Konstruktoren 377
Protected-Elementfunktionen 377
Makro __DELTA . 377
Stream-Makros . 378
Makro DECLARE_STREAMABLE 378
Makro DECLARE_STREAMABLE_FROM_BASE . 378
Makro DECLARE_ABSTRACT_STREAMABLE . 379
Makro DECLARE_STREAMER 379
Makro DECLARE_STREAMER_FROM_BASE . 379
Makro DECLARE_ABSTRACT_STREAMER . 380
Makro DECLARE_CASTABLE 380
Makro DECLARE_STREAMABLE_OPS . 380
Makro DECLARE_STREAMABLE_CTOR 380
Makros IMPLEMENT_STREAMABLE . . 380
Makro IMPLEMENT_STREAMABLE_CLASS . . 381
Makros IMPLEMENT_STREAMABLE_CTOR . 381
Makro IMPLEMENT_STREAMABLE_POINTER . 381
Makro IMPLEMENT_CASTABLE_ID . . . 381
Makros IMPLEMENT_CASTABLE 382
Makro IMPLEMENT_STREAMER 382
Makros IMPLEMENT_ABSTRACT_STREAMABLE . 382
Makro IMPLEMENT_STREAMABLE_FROM_BASE . 382

Kapitel 7 Die C++ Containerklassen . 383

Template TMArrayAsVector 383
Typdefinitionen . 384
Konstruktoren . 384
Public-Funktionstypen 384
Protected-Elementfunktionen 386
Operatoren . 387
Template TMArrayAsVectorIterator 387
Konstruktoren . 388
Public-Elementfunktionen 388
Operatoren . 388
Template TArrayAsVector 388
Konstruktoren . 389
Template TArrayAsVectorIterator 389
Konstruktoren . 389
Template TMIArrayAsVector 389
Typdefinitionen 389
Konstruktoren . 389
Public-Funktionstypen 390
Protected-Elementfunktionen 392
Operatoren . 393
Template TMIArrayAsVectorIterator 393
Konstruktoren . 393
Public-Elementfunktionen 393
Operatoren . 393
Template TIArrayAsVector 394
Konstruktoren . 394
Template TIArrayAsVectorIterator 394
Konstruktoren . 394
Template TMSArrayAsVector 394
Konstruktoren . 394
Template TMSArrayAsVectorIterator 395
Konstruktoren . 395
Template TSArrayAsVector 395
Konstruktoren . 395
Template TSArrayAsVectorIterator 395
Konstruktoren . 395
Template TISArrayAsVector 396
Konstruktoren . 396
Template TISArrayAsVectorIterator 396
Konstruktoren . 396
Template TMISArrayAsVector 396
Konstruktoren . 396
Template TMDDAssociation 397
Konstruktoren . 397
Public-Elementfunktionen 397
Operatoren . 397
Template TDDAssociation 397
Konstruktoren . 398
Template TMDIAssociation 398
Konstruktoren . 398
Public-Elementfunktionen 398
Operatoren . 399
Template TDIAssociation 399
Konstruktoren . 399
Template TMIDAssociation 399

Konstruktoren . 399
Public-Elementfunktionen 400
Operatoren . 400
Template TIDAssociation 400
Konstruktoren . 400
Template TMIIAssociation 401
Konstruktoren . 401
Public-Elementfunktionen 401
Operatoren . 401
Template TIIAssociation 401
Konstruktoren . 402
Template TMBagAsVector 402
Konstruktoren . 402
Typdefinitionen 402
Public-Elementfunktionen 403
Protected-Elementfunktionen 404
Template TMBagAsVectorIterator 404
Konstruktoren . 404
Template TBagAsVector 404
Konstruktoren . 404
Template TBagAsVectorIterator 404
Konstruktoren . 405
Template TMIBagAsVector 405
Typdefinitionen 405
Konstruktoren . 405
Public-Elementfunktionen 405
Template TMIBagAsVectorIterator 406
Konstruktoren . 407
Template TIBagAsVector 407
Konstruktoren . 407
Template TIBagAsVectorIterator 407
Konstruktoren . 407
Template TBinarySearchTreeImp 407
Public-Elementfunktionen 408
Protected-Elementfunktionen 408
Template TBinarySearchTreeIteratorImp . . . 408
Konstruktoren . 409
Public-Elementfunktionen 409
Operatoren . 409
Template TIBinarySearchTreeImp 409
Public-Elementfunktionen 410
Protected-Elementfunktionen 410
Template TIBinarySearchTreeIteratorImp . . 410
Konstruktoren . 411
Public-Elementfunktionen 411
Operatoren . 411
Template TMDequeAsVector 411
Typdefinitionen 412
Konstruktoren . 412
Public-Elementfunktionen 412
Protected-Datenelemente 413
Protected-Elementfunktionen 414
Template TMDequeAsVectorIterator 414
Konstruktoren . 414
Public-Elementfunktionen 414
Operatoren . 414
Template TDequeAsVector 415
Konstruktoren . 415
Template TDequeAsVectorIterator 415
Konstruktoren . 415
Template TMIDequeAsVector 415
Typdefinitionen 415
Konstruktoren . 416
Public-Elementfunktionen 416
Template TMIDequeAsVectorIterator 417
Konstruktoren . 417
Template TIDequeAsVector 417
Konstruktoren . 417
Template TIDequeAsVectorIterator 418
Konstruktoren . 418
Template TMDequeAsDoubleList 418
Typdefinitionen 418
Public-Elementfunktionen 418
Template TMDequeAsDoubleListIterator . . 419
Konstruktoren . 420
Template TDequeAsDoubleList 420
Template TDequeAsDoubleListIterator 420
Konstruktoren . 420
Template TMIDequeAsDoubleList 420
Typdefinitionen 420
Public-Elementfunktionen 421
Template TMIDequeAsDoubleListIterator . 422
Konstruktoren . 422
Template TIDequeAsDoubleList 422
Template TIDequeAsDoubleListIterator . . . 422
Konstruktoren . 422
Template TMDictionaryAsHashTable 423
Protected-Datenelemente 423
Konstruktoren . 423
Public-Elementfunktionen 423

Template TMDictionaryAsHashTableIterator 424
Konstruktoren 424
Public-Elementfunktionen 424
Operatoren 424
Template TDictionaryAsHashTable 425
Konstruktoren 425
Template TDictionaryAsHashTableIterator 425
Konstruktoren 425
Template TMIDictionaryAsHashTable 425
Konstruktoren 425
Public-Elementfunktionen 426
Template TMIDictionaryAsHash TableIterator 426
Konstruktoren 426
Public-Elementfunktionen 427
Operatoren 427
Template TIDictionaryAsHashTable 427
Konstruktoren 427
Template TIDictionaryAsHashTableIterator 428
Konstruktoren 428
Template TDictionary 428
Template TDictionaryIterator 428
Konstruktoren 428
Template TMDoubleListElement 428
Public-Datenelemente 428
Konstruktoren 429
Operatoren 429
Template TMDoubleListImp 429
Typdefinitionen 429
Konstruktoren 430
Public-Elementfunktionen 430
Protected-Datenelemente 431
Protected-Elementfunktionen 431
Template TMDoubleListIteratorImp 431
Konstruktoren 431
Public-Elementfunktionen 431
Operatoren 432
Template TDoubleListImp 432
Konstruktoren 432
Template TDoubleListIteratorImp 432
Konstruktoren 433
Template TMSDoubleListImp 433
Protected-Elementfunktionen 433
Template TMSDoubleListIteratorImp 433
Konstruktoren 433
Template TSDoubleListImp 433
Template TSDoubleListIteratorImp 434
Konstruktoren 434
Template TMIDoubleListImp 434
Typdefinitionen 434
Public-Elementfunktionen 434
Protected-Elementfunktionen 436
Template TMIDoubleListIteratorImp 436
Konstruktoren 436
Public-Elementfunktionen 436
Operatoren 436
Template TIDoubleListImp 436
Template TIDoubleListIteratorImp 437
Konstruktoren 437
Template TMISDoubleListImp 437
Protected-Elementfunktionen 437
Template TMISDoubleListIteratorImp 437
Konstruktoren 438
Template TISDoubleListImp 438
Template TISDoubleListIteratorImp 438
Konstruktoren 438
Template TMHashTableImp 438
Konstruktoren 438
Destruktor 439
Public-Elementfunktionen 439
Template TMHashTableIteratorImp 439
Konstruktoren 440
Destruktor 440
Public-Elementfunktionen 440
Operatoren 440
Template THashTableImp 440
Konstruktoren 441
Template THashTableIteratorImp 441
Konstruktoren 441
Template TMIHashTableImp 441
Konstruktoren 441
Public-Elementfunktionen 441
Template TMIHashTableIteratorImp 442
Konstruktoren 442
Public-Elementfunktionen 442
Operatoren 443
Template TIHashTableImp 443
Konstruktoren 443
Template TIHashTableIteratorImp 443
Konstruktoren 443

Template TMListElement 444
Public-Datenelemente 444
Konstruktoren . 444
Operatoren . 444
Template TMListImp 444
Typdefinitionen 445
Konstruktoren . 445
Public-Elementfunktionen 445
Protected-Datenelemente 446
Protected-Elementfunktionen 446
Template TMListIteratorImp 446
Konstruktoren . 446
Public-Elementfunktionen 447
Operatoren . 447
Template TListImp 447
Template TListIteratorImp 447
Konstruktoren . 447
Template TMSListImp 448
Template TMSListIteratorImp 448
Konstruktoren . 448
Template TSListImp 448
Template TSListIteratorImp 448
Template TMIListImp 448
Typdefinitionen 449
Public-Elementfunktionen 449
Protected-Elementfunktionen 449
Template TMIListIteratorImp 450
Konstruktoren . 450
Public-Elementfunktionen 450
Operatoren . 450
Template TIListImp 450
Template TIListIteratorImp 450
Konstruktoren . 451
Template TMISListImp 451
Public-Elementfunktionen 451
Template TMISListIteratorImp 451
Konstruktoren . 451
Template TISListImp 451
Template TISListIteratorImp 452
Konstruktoren . 452
Template TMQueueAsVector 452
Konstruktoren . 452
Public-Elementfunktionen 452
Template TMQueueAsVectorIterator 453
Konstruktoren . 453
Template TQueueAsVector 453
Konstruktoren . 453
Template TQueueAsVectorIterator 454
Konstruktoren . 454
Template TMIQueueAsVector 454
Konstruktoren . 454
Public-Elementfunktionen 454
Template TMIQueueAsVectorIterator 455
Konstruktoren . 455
Template TIQueueAsVector 455
Konstruktoren . 455
Template TIQueueAsVectorIterator 456
Konstruktoren . 456
Template TMQueueAsDoubleList 456
Public-Elementfunktionen 456
Template TMQueueAsDoubleListIterator . . 457
Konstruktoren . 457
Template TQueueAsDoubleList 457
Template TQueueAsDoubleListIterator 457
Konstruktoren . 457
Template TMIQueueAsDoubleList 458
Public-Elementfunktionen 458
Template TMIQueueAsDoubleListIterator . 459
Konstruktoren . 459
Template TIQueueAsDoubleList 459
Template TIQueueAsDoubleListIterator . . . 459
Konstruktoren . 459
Template TQueue 459
Template TQueueIterator 460
Template TMSetAsVector 460
Konstruktoren . 460
Public-Elementfunktionen 460
Template TMSetAsVectorIterator 460
Konstruktoren . 460
Template TSetAsVector 461
Konstruktoren . 461
Template TSetAsVectorIterator 461
Konstruktoren . 461
Template TMISetAsVector 461
Konstruktoren . 461
Public-Elementfunktionen 462
Template TMISetAsVectorIterator 462
Konstruktoren . 462
Template TISetAsVector 462
Konstruktoren . 462

Template TISetAsVectorIterator 462
Konstruktoren 463
Template TSet 463
Template TSetIterator 463
Template TMStackAsVector 463
Typdefinitionen 463
Konstruktoren 463
Public-Elementfunktionen 464
Template TMStackAsVectorIterator 465
Konstruktoren 465
Template TStackAsVector 465
Konstruktoren 465
Template TStackAsVectorIterator 465
Konstruktoren 465
Template TMIStackAsVector 466
Typdefinitionen 466
Konstruktoren 466
Public-Elementfunktionen 466
Template TMIStackAsVectorIterator 467
Konstruktoren 467
Template TIStackAsVector 467
Konstruktoren 468
Template TIStackAsVectorIterator 468
Konstruktoren 468
Template TMStackAsList 468
Template TMStackAsListIterator 468
Konstruktoren 468
Template TStackAsList 469
Template TStackAsListIterator 469
Konstruktoren 469
Template TMIStackAsList 469
Template TMIStackAsListIterator 469
Konstruktoren 469
Template TIStackAsList 470
Template TIStackAsListIterator 470
Konstruktoren 470
Template TStack 470
Template TStackIterator 470
Template TMVectorImp 470
Typdefinitionen 471
Konstruktoren 471
Public-Elementfunktionen 471
Operatoren 472
Protected-Datenelemente 473
Protected-Elementfunktionen 473
Template TMVectorIteratorImp 473
Konstruktoren 473
Public-Elementfunktionen 473
Operatoren 474
Template TVectorImp 474
Konstruktoren 474
Template TVectorIteratorImp 474
Konstruktoren 475
Template TMCVectorImp 475
Konstruktoren 475
Public-Elementfunktionen 475
Protected-Datenelemente 476
Protected-Elementfunktionen 476
Template TMCVectorIteratorImp 476
Konstruktoren 476
Template TCVectorImp 477
Konstruktoren 477
Template TCVectorIteratorImp 477
Konstruktoren 477
Template TMSVectorImp 478
Konstruktoren 478
Template TMSVectorIteratorImp 478
Konstruktoren 478
Template TSVectorImp 478
Konstruktoren 479
Template TSVectorIteratorImp 479
Konstruktoren 479
Template TMIVectorImp 479
Typdefinitionen 479
Konstruktoren 480
Public-Funktionstypen 480
Operatoren 481
Template TMIVectorIteratorImp 481
Konstruktoren 481
Public-Elementfunktionen 481
Operatoren 481
Template TIVectorImp 482
Konstruktoren 482
Template TIVectorIteratorImp 482
Konstruktoren 482
Template TMICVectorImp 482
Konstruktoren 483
Public-Elementfunktionen 483
Protected-Elementfunktionen 483
Template TMICVectorIteratorImp 483

Konstruktoren ... 483
Template TICVectorImp ... 484
Konstruktoren ... 484
Template TICVectorIteratorImp ... 484
Konstruktoren ... 484
Template TMISVectorImp ... 484
Konstruktoren ... 485
Template TMISVectorIteratorImp ... 485
Konstruktoren ... 485
Template TISVectorImp ... 485
Konstruktoren ... 485
Template TISVectorIteratorImp ... 486
Konstruktoren ... 486
Klasse TShouldDelete ... 486
Public-Datenelemente ... 486
Konstruktoren ... 486
Public-Elementfunktionen ... 487
Protected-Elementfunktionen ... 487

Kapitel 8 Die mathematischen C++ Klassen ... 489
Klasse bcd ... 489
Public-Konstruktoren ... 490
Friend-Funktionen ... 492
Klasse complex ... 492
Public-Konstruktoren ... 492
Friend-Funktionen ... 493

Kapitel 9 Diagnose-Makros ... 497
Standard-Diagnose-Makros ... 498
Erweiterte Diagnose-Makros ... 499
Ausgabe von Meldungen ... 502
Makrosteuerung zur Laufzeit ... 502

Kapitel 10 Laufzeitunterstützung ... 503
Klasse Bad_cast ... 503
Klasse Bad_typeid ... 503
Funktion set_new_handler ... 503
Funktion set_terminate ... 505
Funktion set_unexpected ... 505
Funktion terminate ... 506
Klasse Type_info ... 506
Public-Konstruktoren ... 506
Operatoren ... 507
Public-Elementfunktionen ... 507
Funktion unexpected ... 507
Klasse xalloc ... 508
Public-Konstruktoren ... 508
Public-Elementfunktionen ... 508
Klasse xmsg ... 508
Public-Konstruktoren ... 508
Public-Elementfunktionen ... 508

Kapitel 11 Die C++ Utility-Klassen ... 511
Klasse TDate ... 511
Typdefinitionen ... 511
Public-Konstruktoren ... 512
Public-Elementfunktionen ... 512
Protected-Elementfunktionen ... 514
Operatoren ... 514
Struktur TFileStatus ... 516
Klasse TFile ... 516
Public-Datenelemente ... 516
Konstruktoren ... 517
Public-Elementfunktionen ... 517
Klasse string ... 519
Typdefitionen ... 519
Konstruktoren ... 519
Destruktor ... 520
Public-Elementfunktionen ... 520
Protected-Elementfunktionen ... 527
Operatoren ... 528
Verwandte globale Funktionen und Operatoren ... 530
Klasse TSubString ... 531
Public-Elementfunktionen ... 531
Protected-Elementfunktionen ... 532
Operatoren ... 532
Klasse TCriticalSection ... 533
Konstruktoren ... 533
Destruktor ... 533
Klasse TCriticalSection::Lock ... 533
Konstruktoren ... 534
Destruktor ... 534
Klasse TMutex ... 534
Konstruktoren ... 535
Destruktor ... 535

Operatoren . 535
Klasse TMutex::Lock 535
Konstruktoren . 535
Public-Elementfunktionen 535
Klasse TSync . 535
Protected-Konstruktoren 536
Protected-Operatoren 536
Klasse TSync::Lock 537
Konstruktoren . 537
Destruktor . 537
Klasse TThread . 537
Typdefinitionen 538
Protected-Konstruktoren 538
Protected-Destruktor 539
Public-Elementfunktionen 539
Protected-Elementfunktionen 540
Protected-Operatoren 540
Klasse TThread::TThreadError 540
Typdefinitionen 540
Public-Elementfunktionen 541
Typdefinitionen für TTime 541
Klasse TTime . 541
Protected-Datenelemente 541
Konstruktoren . 542
Public-Elementfunktionen 542
Protected-Elementfunktionen 543
Operatoren . 543

Tabellen

1.1 Standard-Laufzeitblibiotheken 23
1.2 Zusammenfassung der statische Laufzeitbibliotheken 23
1.3 Dynamische Linkbibliotheken, Zusammenfassung 25
1.4 Veraltete globale Variablen 35
1.5 Veraltete Funktionsnamen 35
3.1 Locale-Einstellungen für Währungs- und numerische Formate 178
3.2 Diese Meldungen werden unter Win 16 und Win 32 erzeugt. 210
3.3 Diese Meldungen werden nur unter Win 16 und Win 32 erzeugt. 211

Bilder

6.1 Klassenhierarchie
für persistente Streams . 363

Einführung

Das vorliegende Handbuch enthält die Definitionen aller Borland C++ Klassen, der nichtprivaten Klassenelemente, der Bibliotheksfunktionen, der globalen Variablen und der global definierten Datentypen.

Wenn Sie mit der C- oder C++ Programmierung erst beginnen, dann sollten Sie zunächst den Einführungsteil des *Benutzerhandbuchs* lesen.

Dieses Handbuch ist folgendermaßen aufgebaut:

Kapitel 1: Bibliotheksreferenz enthält einen Überblick der Borland C++ Bibliotheksfunktionen und Header-Dateien. Die statischen und dynamischen Linkbibliotheken werden beschrieben, und die Header-Dateien werden aufgelistet. Die Bibliotheksfunktionen werden dabei anhand ihrer Aufgaben zu Gruppen zusammengefaßt.

Kapitel 2: Die Funktion main beschreibt die Argumente, die an die Funktion *main* übergeben werden (einschließlich Argumenten, die Jokerzeichen enthalten). Weiterhin werden einige Beispielprogramme vorgestellt, und die Pascal-Aufrufkonvention sowie der Rückgabewert von *main* werden beschrieben.

Kapitel 3: Bibliotheksfunktionen listet alle Borland C++ Bibliotheksfunktionen in alphabetischer Reihenfolge auf. Zu jedem Eintrag gehört die Syntax, ein Hinweis auf die Portabilität, eine praxisnahe Beschreibung, der Rückgabewert der Funktion und ein Verweis auf verwandte Funktionen.

Kapitel 4: Globale Variablen definiert und beschreibt die globalen Variablen von Borland C++. Benutzen Sie diese Variablen (Zeit- und Datumsangaben, Fehlermeldungen, Stack-Größe etc.), und Sie können bei Ihrer Programmierarbeit viel Zeit sparen.

Kapitel 5: Die C++ Klassen für Ein-/Ausgabe-Streams beschreibt die Klassen, die die Ein-/Ausgabe in C++ Programmen unterstützen.

Kapitel 6: Klassen und Makros für persistente Streams beschreibt die Klassen und Makros für persistente Streams.

Kapitel 7: Die C++ Containerklassen stellt die Containerklassen von Borland C++ vor, wie etwa Array, Stack und verkettete Liste.

Kapitel 8: Die mathematischen C++ Klassen beschreibt, wie die Klassen *bcd* und *complex* verwendet werden.

Kapitel 9: Diagnose-Makros beschreibt die Klassen und Makros, mit denen Objekte untersucht werden können.

Kapitel 10: Laufzeitunterstützung beschreibt Funktionen und Klassen, mit deren Hilfe Sie das Laufzeitverhalten Ihres Programms in den Fällen steuern können, wenn der Speicherplatz knapp wird oder andere Fehler auftreten.

Kapitel 11: Die C++ Utility-Klassen beschreibt die C++ Klassen *date*, *string* und *time*.

K A P I T E L 1

Bibliotheksreferenz

Dieses Kapitel beinhaltet einen Überblick über die Borland C++ Bibliotheksroutinen und Header-Dateien. Die Bibliotheksroutinen setzen sich aus Klassen, Funktionen und Makros zusammen, die Sie innerhalb Ihrer C- und C++ Programme aufrufen und die eine große Vielfalt von Aufgaben übernehmen. Dazu gehören beispielsweise Ein-/Ausgabe, String- und Dateibehandlung, Speicherplatzreservierung, Prozeßsteuerung, Datenkonvertierung, mathematische Berechnungen und vieles mehr.

Dieses Kapitel enthält die folgenden Informationen:

- Die Namen der statischen und dynamischen Linkbibliotheken, der zugehörigen Dateien und Unterverzeichnisse, die sich in den Unterverzeichnissen LIB und BIN befinden, werden aufgelistet, und ihre Verwendung wird beschrieben.
- Sie erfahren, warum es sinnvoll sein kann, den Quelltext der Borland C++ Laufzeitbibliothek zu besitzen.
- Die Header-Dateien werden aufgelistet und beschrieben.
- Die Bibliotheksroutinen werden – nach Aufgabenbereichen in Kategorien eingeteilt – aufgelistet.

Warum der Quelltext der Laufzeitbibliothek wichtig ist

Es gibt einige gute Gründe dafür, warum Ihnen der Quelltext der Laufzeitbibliothek von Nutzen sein kann:

- Sie wollen eine bestimmte Funktion entwickeln, die einer Borland C++ Funktion zwar ähnelt aber nicht mit ihr identisch ist. Steht Ihnen der Quelltext der Laufzeitbibliothek zur Verfügung, so können Sie die Bibliotheksfunktion Ihren Bedürfnissen anpassen und die Neuentwicklung einer eigenen Funktion vermeiden.
- Beim Debuggen möchten Sie mitunter gern mehr über die Interna einer Bibliotheksfunktion erfahren.
- Sie können die führenden Unterstriche von C-Symbolen entfernen, wenn Sie dies möchten und Zugriff auf die Bibliotheks-Quelltexte haben.
- Sie profitieren von der Beschäftigung mit dem kompakten, professionell geschriebenen Quelltext der Laufzeitbibliothek.

All dieses – und noch mehr – könnten gute Gründe für Sie sein, den Quelltext der Laufzeitbibliothek haben zu wollen. Da Borland das Konzept einer "offenen Architektur" vertritt, stellen wir Ihnen den Quelltext der Laufzeitbibliothek zur Verfügung. Dazu müssen Sie lediglich das Bestellformular ausfüllen, das Ihrem Borland C++ Paket beiliegt, und wir senden Ihnen den Quelltext der Borland C++ Laufzeitbibliothek zu.

Die Laufzeitbibliotheken

Die Laufzeitbibliotheken lassen sich in statische (OBJ und LIB) sowie dynamische (DLL) Bibliotheken unterteilen. Die verschiedenen Bibliotheksversionen werden in getrennten Verzeichnissen installiert und im folgenden auch in getrennten Tabellen beschrieben.

Im *Object Windows Referenzhandbuch* finden Sie Informationen über die speziellen Bibliotheken und DLLs für Object Windows.

Es gibt verschiedene Versionen der Laufzeitbibliotheken, beispielsweise verschiedene Versionen für jedes Speichermodell, für Diagnosezwecke und für 16- oder 32-Bit Programme.

Weiterhin existieren Bibliotheken, die mathematischen Funktionen, Containerklassen, Object Windows Entwicklungen oder internationale Anwendungen unterstützen.

Hier einige Hinweise zur Auswahl der Laufzeitbibliotheken:

- Bibliotheken für segmentierte Speichermodelle werden nur in 16-Bit Programmen unterstützt. Die Speichermodelle Tiny und Huge werden nicht unterstützt.
- Bei 32-Bit Programmen wird nur das Speichermodell Flat unterstützt.

- Bei 32-Bit Programmen werden Anwendungen für Textbildschirme und für grafische Benutzeroberflächen (GUI) von verschiedenen Bibliotheken unterstützt.
- Multi-Thread-Anwendungen werden nur in 32-Bit Programmen unterstützt.

Die statischen Bibliotheken

Die statische Version (OBJ und LIB) der Borland C++ Laufzeitbibliothek befindet sich im Unterverzeichnis LIB Ihrer Installation. Bei den folgenden Dateinamen vertritt das Fragezeichen ("?") jeweils eines der vier von Borland unterstützten Speichermodelle. Jedes Modell besitzt seine eigene Bibliotheks- und Mathematikdatei, die die Version der Routine enthält, die für das entsprechende Speichermodell entwickelt wurde. Im Kapitel 8 des *Programmierhandbuchs* finden Sie Einzelheiten zu Speichermodellen.

Die folgende Tabelle enthält die Standard-Laufzeitbibliotheken für jeden Compiler. Im *Benutzerhandbuch* erfahren Sie näheres zum Compilieren und Linken.

Tabelle 1.1: Standard-Laufzeitblibiotheken

Compiler	Anwendungstyp	Standardbibliothek
BCC.EXE	16-Bit Windows	C0DS.OBJ, CWS.LIB, MATHWS.LIB, IMPORT.LIB
BCC32.EXE	Win32, Win32s	C0W32.OBJ, CW32.LIB, IMPRTW32.LIB
BCW.EXE	wie bei BCC.EXE	wie bei BCC.EXE
BCWS32.EXE	wie bei BCC32.EXE	wie bei BCC32.EXE

In der folgenden Tabelle werden die Namen der statischen Bibliotheken von Borland C++ zusammen mit dem Betriebssystem, unter dem sie verfügbar sind, aufgelistet. Im Kapitel 3 des *Benutzerhandbuchs* finden Sie Informationen über Linker, Linkeroptionen, Anforderungen und Auswahl von Bibliotheken.

Tabelle 1.2: Zusammenfassung der statischen Laufzeitbibliotheken

Dateiname	Anwendungstyp	Verwendung
Verzeichnis BC4\LIB		
BIDSDI.LIB	Win 16	16-Bit dynamische BIDS Diagnose-Importbibliothek für BIDS40D.DLL
BIDSI.LIB	Win 16	16-Bit dynamische BIDS Importbibliothek für BIDS40D.DLL
BIDSF.LIB	Win32s, Win32	32-Bit BIDS Bibliothek
BIDSDF.LIB	Win32s, Win32	32-Bit BIDS Diagnose-Bibliothek
BIDSFI.LIB	Win32s, Win32	32-Bit dynamische BIDS Importbibliothek für BIDS40.DLL
BIDSDFI.LIB	Win32s, Win32	32-Bit dynamische BIDS Diagnose-Importbibliothek für BIDS40DF.DLL
BIDSDB?.LIB	Win 16	16-Bit BIDS Diagnose-Bibliothek

BIDS?.LIB	Win 16	16-Bit BIDS Bibliothek
BWCC.LIB	Win 16	16-Bit Importbibliothek für BWCC.DLL
BWCC32.LIB	Win32s, Win32	32-Bit Importbibliothek für BWCC32.DLL
C0D32.OBJ	Win32s, Win32	32-Bit DLL Startmodul
C0D?.OBJ	Win 16	16-Bit DLL Startmodul
C0W32.OBJ	Win32s, Win32	32-Bit GUI-EXE Startmodul
C0W?.OBJ	Win 16	16-Bit EXE Startmodul
C0X32.OBJ	Win 32	32-Bit Textmodus EXE Startmodul
CRTDLL.LIB	Win 16	16-Bit dynamische Importbibliothek für BC40RTL.DLL
CW32.LIB	Win32s, Win32	32-Bit GUI Single-Thread Bibliothek
CW?.LIB	Win 16	16-Bit Bibliothek
CW32I.LIB	Win32s, Win32	32-Bit Single-Thread, GUI, dynamische RTL Importbibliothek für CW32.DLL
CW32MT.LIB	Win32	32-Bit GUI Multi-Thread Bibliothek
CW32MTI.LIB	Win32	32-Bit Multi-Thread, GUI, dynamische RTL Importbibliothek für CW32MT.DLL
IMPORT.LIB	Win 16	16-Bit Importbibliothek für Windows 3.1
IMPORT32.LIB	Win32s, Win32	32-Bit Importbibliothek; Verwendung mit IMPRTW32.LIB
MATHW?.LIB	Win 16	16-Bit Mathematikbliotheken
W32SUT16.LIB	Win 16	16-Bit universelle Thunk-Bibliothek
W32SUT32.LIB	Win32s	32-Bit universelle Thunk-Bibliothek
OBSOLETE.LIB	Win 16, Win32s, Win32	Stellt veraltete globale Variablen zur Verfügung

Verzeichnis BC4\LIB\16BIT

FILES.C	Win 16	Erhöhung der Anzahl verfügbarer Datei-Handles
FILES2.C	Win 16	Erhöhung der Anzahl verfügbarer Datei-Handles
MATHERR.C	Win 16	Beispiel einer benutzerdefinierten Routine zur Behandlung von Gleitkommafehlern für die Datentypen **float** und **double**
MATHERRL.C	Win 16	Beispiel einer benutzerdefinierten Routine zur Behandlung von Gleitkommafehlern für den Datentyp **long double**

Verzeichnis BC4\LIB\32BIT

FILES.C	Win32s, Win32	Erhöhung der Anzahl verfügbarer Datei-Handles
FILES2.C	Win32s, Win32	Erhöhung der Anzahl verfügbarer Datei-Handles
FILEINFO.OBJ	Win32s, Win32	Übergabe der Informationen über offene Dateien an Kindprozesse
GP.OBJ	Win32s, Win32	Ausgabe des Registerinhalts bei Auftreten eines Fehlers
MATHERR.C	Win32s, Win32	Beispiel einer benutzerdefinierten Routine zur Behandlung von Gleitkommafehlern für die Datentypen **float** und **double**

MATHERRL.C	Win32s, Win32	Beispiel einer benutzerdefinierten Routine zur Behandlung von Gleitkommafehlern für den Datentyp **long double**
WILDARGS.OBJ	Win32	Umwandlung von Jokerzeichen-Parametern in ein Array von Argumenten der Funktion *main* bei Textbildschirmanwendungen
Verzeichnis BC4\LIB\STARTUP		
BUILD-C0.BAT	Win 16	Batch-Datei zum Aufbau von C0D?.OBJ, C0F?.OBJ und C0W?.OBJ
C0D.ASM	Win 16	Quelltext für C0D?.OBJ
C0W.ASM	Win 16	Quelltext für C0W?.OBJ
RULES.ASI	Win 16	Assembler-Regeln für C0D.ASM und C0W.ASM

Die dynamischen Linkbibliotheken (DLL)

Die Version der Laufzeitbibliothek, die zum dynamischen Linken geeignet ist (DLL), befindet sich im Verzeichnis BIN der Installation. Es gibt verschiedene Versionen von DLL-Bibliotheken, z.B. für Diagnosezwecke, für 16- und für 32-Bit Programme sowie Versionen, die Multi-Thread-Anwendungen unterstützen.

In der 16-Bit Version wird nur die DLL für das Speichermodell Large zur Verfügung gestellt. Von einer 16-Bit DLL wird kein weiteres Speichermodell unterstützt.

Die folgende Tabelle enthält die Namen der dynamischen Linkbibliotheken von Borland C++ , ihre Verwendung sowie das Betriebssystem, unter dem die jeweilige Bibliothek verfügbar ist. Im Kapitel 8 des *Benutzerhandbuchs* finden Sie weitere Informationen über Linker, Linkeroptionen, Anforderungen und Auswahl von Bibliotheken.

Tabelle 1.3: Dynamische Linkbibliotheken, Zusammenfassung

Verzeichnis BC4\BIN

Dateiname	**Anwendungstyp**	**Verwendung**
BC40RTL.DLL	Win 16	16-Bit, Speichermodell Large
BIDS40.DLL	Win 16	16-Bit, BIDS
BIDS40D.DLL	Win 16	16-Bit, BIDS Diagnosebibliothek
BIDS40F.DLL	Win32s, Win32	32-Bit, BIDS
BIDS40DF.DLL	Win32s, Win32	32-Bit, BIDS Diagnosebibliothek
CW32.DLL	Win32s, Win32	32-Bit, Single-Thread
CW32MT.DLL	Win 32	32-Bit, Multi-Thread
LOCALE.BLL	Win 16, Win32s, Win32	Locale-Bibliothek

Die Borland C++ Header-Dateien

C++ Header-Dateien und von ANSI-C definierte Header-Dateien, sind durch Fußnoten gekennzeichnet.

Header-Dateien enthalten die Deklaration der Funktionsprototypen der Bibliotheksfunktionen. Auch Datentypen und symbolische Konstanten, die in den Bibliotheksfunktionen verwendet werden, sind hier definiert. Schließlich sind auch globale Variablen, die von Borland C++ und den Bibliotheksfunktionen definiert werden, in den Header-Dateien enthalten. Die Borland C++ Bibliothek folgt dem ANSI-C Standard bezüglich der Namen der Header-Dateien und ihres Inhalts.

Header-Datei	Beschreibung
alloc.h	Deklariert Funktionen zur Speicherverwaltung (Reservierung, Freigabe etc.)
assert.h[1]	Definiert das Debug-Makro *assert*.
bcd.h[2]	Deklariert die C++ Klasse *bcd*, die überladenen Operatoren von *bcd* sowie die zugehörigen mathematischen Funktionen.
bios.h	Deklariert verschiedene Funktionen, die zum Aufruf von IBM PC ROM BIOS-Routinen verwendet werden.
checks.h[2]	Definiert die Diagnose-Makros PRECONDITION, WARN und TRACE.
complex.h[2]	Deklariert die mathematische C++ Klasse *complex*.
conio.h	Deklariert verschiedene Funktionen, die beim Aufruf der Betriebssystemroutinen zur Ein-/Ausgabe auf Textbildschirmen verwendet werden. Die in dieser Header-Datei definierten Funktionen können nicht in GUI-Anwendungen benutzt werden.
constrea.h[2]	Deklariert C++ Klassen und Methoden zur Ausgabe auf Textbildschirmen.
cstring.h[2]	Deklariert die ANSI C++ Strings.
ctype.h[1]	Enthält Informationen, die von den Makros zur Zeichenklassifizierung und -konvertierung benutzt werden (wie zum Beispiel *isalpha* und *toascii*).
dir.h	Enthält Strukturen, Makros und Funktionen für den Umgang mit Verzeichnissen und Pfadnamen.
direct.h	Definiert Strukturen, Makros und Funktionen für den Umgang mit Verzeichnissen und Pfadnamen.
dirent.h	Deklariert Funktionen und Strukturen für Verzeichnis-Operationen entsprechend dem POSIX Standard.
dos.h	Enthält verschiedene Konstanten und Deklarationen, die für spezielle DOS- und 80x86-Aufrufe benötigt werden.
errno.h[1]	Definiert Konstanten für die Fehlercodes.
except.h[2]	Deklariert ANSI C++ Exceptions.
excpt.h	Deklariert C Exceptions.
fcntl.h	Definiert symbolische Konstanten, die für die Bibliotheksfunktion *open* benötigt werden.
float.h[1]	Enthält Parameter für Gleitkommaroutinen.
fstream.h[2]	Deklariert die C++ Stream-Klassen für die Dateiein- und ausgabe.
generic.h	Enthält Makros zur Deklaration generischer Klassen.
io.h	Enthält Strukturen und Deklarationen für Ein-/Ausgaberoutinen.

iomanip.h[2]	Deklariert E/A-Manipulatoren der C++ Streams und enthält Templates zur Erzeugung parametrisierter Manipulatoren.
iostream.h[2]	Deklariert die grundlegenden C++ Stream-Ein-/Ausgaberoutinen.
limits.h[1]	Enthält Systemparameter, Informationen über Beschränkungen zum Zeitpunkt der Compilierung sowie Wertebereiche von ganzzahligen Größen.
locale.h[1]	Deklariert Funktionen, die länder- und sprachenabhängige Informationen bereitstellen.
sys\locking.h	Definitionen für den Parameter *mode* der Funktion *locking*.
malloc.h	Funktionen und Variablen zur Speicherverwaltung
math.h[1]	Deklariert Prototypen für mathematische Funktionen und Fehlerbehandlungsroutinen.
mem.h	Deklariert Funktionen zur Speicherverwaltung (viele davon sind auch in string.h definiert).
memory.h	Fuktionen zur Speicherverwaltung.
new.h[2]	Ermöglicht den Zugriff auf *_new_handler* und *set_new_handler*.
process.h	Enthält Strukturen und Deklarationen für die Funktionsgruppen *spawn...* und *exec...* .
ref.h[2]	Unterstützt das Zählen von Referenzen und wird zusammen mit Strings verwendet.
regexp.h[2]	Implementiert die Suche nach regulären Ausdrücken.
search.h	Deklariert Such- und Sortierfunktionen.
setjmp.h[1]	Definiert den Typ *jmp_buf*, der von den Funktionen *longjmp* und *setjmp* verwendet wird. Deklariert die Funktionen *longjmp* und *setjmp*.
share.h	Definiert Parameter für Funktionen, die einen gemeinsamen Dateizugriff unterstützen.
signal.h[1]	Enthält Konstanten und Deklarationen für die Funktionen *signal* und *raise*.
stdarg.h[1]	Definiert Makros, mit denen Funktionen Argumentlisten mit variabler Argumentanzahl lesen können (wie beispielsweise bei *vprintf*, *vscanf* etc.).
stddef.h[1]	Definiert verschiedene allgemeine Datentypen und Makros.
stdio.h[1]	Definiert Typen und Makros, die für die im Kernigham & Ritchie definierte und unter UNIX System V erweiterte Standard-Ein-/Ausgabe benötigt werden. Definiert die vordefinierten Ein-/Ausgabe-Streams *stdin*, *stdout*, *stdprn* und *stderr*. Deklariert Ein-/Ausgabe-Funktionen für Streams.
stdiostr.h[2]	Deklariert die C++ (Version 2.0) Stream-Klassen zur Verwendung mit den FILE-Strukturen von stdio. Bei Neuentwicklungen sollten Sie iostream.h einsetzen.
stdlib.h[1]	Deklariert mehrere, allgemein benutzte Routinen zur Konvertierung, zum Suchen und Sortieren sowie verschiedene weitere.
string.h[1]	Deklariert mehrere Routinen zur String- und Speichermanipulation.
strstrea.h[2]	Deklariert die C++ Stream-Klassen für Byte-Arrays im Speicher.
sys\stat.h	Definiert symbolische Konstanten, die beim Öffnen und Erzeugen von Dateien verwendet werden.
time.h[1]	Definiert eine Struktur, die von den Routinen zur Zeitkonvertierung (*asctime*, *localtime* und *gmtime*) gefüllt wird. Definiert einen Typ für die Routinen *ctime, localtime* und *stime*. Enthält die Prototypen dieser Routinen.
sys\timeb.h	Deklariert die Funktion *ftime* und die Struktur *timeb*, die von *ftime* zurückgegeben wird.
sys\types.h	Deklariert den Typ *time_t*, der von den Zeitfunktionen benutzt wird.

typeinfo.h[2]	Enthält Deklarationen für die ANSI C++ Typidentifizierung zur Laufzeit (RTTI - run time typ identification).
utime.h	Deklariert die Funktion *utime* sowie die von ihr zurückgegebene Struktur *utimebuf*.
values.h	Definiert wichtige Konstanten, wie beispielsweise über Maschinenabhängigkeiten. Wird für die Kompatibilität mit UNIX System V benötigt.
varargs.h	Definitionen für den Zugriff auf Parameter von Funktionen, die eine variable Parameteranzahl übernehmen können. Wird aus Kompatibilitätsgründen zu UNIX bereitgestellt. Bei Neuentwicklungen sollten Sie stdarg.h verwenden.

[1] Definiert von ANSI C.
[2] C++ Header-Dateien.

Bibliotheksroutinen nach Kategorien geordnet

Die Borland C++ Bibliotheksroutinen sind für unterschiedliche Aufgaben gedacht. In diesem Abschnitt werden diese Routinen aufgelistet und dabei entsprechend ihrer Verwendung zu Kategorien zusammengefaßt.

Routinen mit C++ Prototypen

Verschiedene Routinen, die in diesem Buch beschrieben werden, werden an mehreren Stellen deklariert. Sie müssen daher den für Ihr Programm geeigneten Prototyp auswählen. Die Ursache für mehrfache Prototypen liegt darin, daß sowohl die ursprüngliche C-Implementation als auch die strengere und teilweise auch unterschiedliche C++ Funktionsdeklaration unterstützt werden. So haben beispielsweise einige Funktionen zur Stringverarbeitung mehrere Prototypen, da Borland C++ neben den von ANSI-C spezifizierten Prototypen auch die Prototypen unterstützt, die dem ANSI C++ Entwurf entsprechen.

getvect	(dos.h)	*strchr*	(string.h)
max	(stdlib.h)	*strpbrk*	(string.h)
memchr	(string.h)	*strrchr*	(string.h)
min	(stdlib.h)	*strstr*	(string.h)
setvect	(dos.h)		

Klassifizierungsroutinen

Diese Routinen klassifizieren ein ASCII-Zeichen als Buchstaben, Steuerzeichen, Interpunktionszeichen, Großbuchstaben etc.

isalnum	(ctype.h)	*islower*	(ctype.h)
isalpha	(ctype.h)	*isprint*	(ctype.h)
isascii	(ctype.h)	*ispunct*	(ctype.h)
iscntrl	(ctype.h)	*isspace*	(ctype.h)
isdigit	(ctype.h)	*isupper*	(ctype.h)
isgraph	(ctype.h)	*isxdigit*	(ctype.h)

Routinen zur Ein-/Ausgabe in Textbildschirmanwendungen

Mit diesen Routinen kann Text auf dem Bildschirm ausgegeben beziehungsweise von der Tastatur eingelesen werden. In GUI-Anwendungen können diese Routinen nicht eingesetzt werden.

cgets	(conio.h)	*movetext*	(conio.h)
clreol	(conio.h)	*normvideo*	(conio.h)
clrscr	(conio.h)	*putch*	(conio.h)
cprintf	(conio.h)	*puttext*	(conio.h)
cputs	(conio.h)	*_setcursortype*	(conio.h)
delline	(conio.h)	*textattr*	(conio.h)
getpass	(conio.h)	*textbackground*	(conio.h)
gettext	(conio.h)	*textcolor*	(conio.h)
gettextinfo	(conio.h)	*textmode*	(conio.h)
gotoxy	(conio.h)	*ungetc*	(stdio.h)
highvideo	(conio.h)	*wherex*	(conio.h)
insline	(conio.h)	*wherey*	(conio.h)
lowvideo	(conio.h)	*window*	(conio.h)

Konvertierungsroutinen

Mit Hilfe dieser Funktionen können Zeichen und Strings von der Textdarstellung in verschiedene numerische Darstellungen (Gleitkomma, Integer, **long**) konvertiert werden und umgekehrt. Weiterhin sind Konvertierungen von Groß- in Kleinschreibung und zurück möglich.

atof	(stdlib.h)	*strtol*	(stdlib.h)
atoi	(stdlib.h)	*_strtold*	(stdlib.h)
atol	(stdlib.h)	*strtoul*	(stdlib.h)
ecvt	(stdlib.h)	*toascii*	(ctype.h)
fcvt	(stdlib.h)	*_tolower*	(ctype.h)
gcvt	(stdlib.h)	*tolower*	(ctype.h)
itoa	(stdlib.h)	*_toupper*	(ctype.h)
ltoa	(stdlib.h)	*toupper*	(ctype.h)
strtod	(stdlib.h)	*ultoa*	(stdlib.h)

Diagnoseroutinen

Diese Routinen unterstützen die Behandlung von Fehlerfällen.

assert	(assert.h)	*perror*	(errno.h)
CHECK	(checks.h)	*PRECONDITION*	(checks.h)
_matherr	(math.h)	*TRACE*	(checks.h)
_matherrl	(math.h)	*WARN*	(checks.h)

Routinen zur Verzeichnisverwaltung

Diese Routinen arbeiten mit Verzeichnissen und Pfadnamen.

chdir	(dir.h)	*_getdcwd*	(direct.h)
_chdrive	(direct.h)	*getdisk*	(dir.h)
closedir	(dirent.h)	*_makepath*	(stdlib.h)
_dos_findfirst	(dos.h)	*mkdir*	(dir.h)
_dos_findnext	(dos.h)	*mktemp*	(dir.h)
_dos_getdiskfree	(dos.h)	*opendir*	(dirent.h)
_dos_getdrive	(dos.h)	*readdir*	(dirent.h)

_dos_setdrive	(dos.h)	*rewinddir*	(dirent.h)
findfirst	(dir.h)	*rmdir*	(dir.h)
findnext	(dir.h)	*_searchenv*	(stdlib.h)
fnmerge	(dir.h)	*searchpath*	(dir.h)
fnsplit	(dir.h)	*_searchstr*	(stdlib.h)
_fullpath	(stdlib.h)	*setdisk*	(dir.h)
getcurdir	(dir.h)	*_splitpath*	(stdlib.h)
getcwd	(dir.h)		

EasyWin-Routinen

Diese Routinen können in EasyWin-Programme portiert werden, stehen jedoch in 16-Bit Windows-Programmen nicht zur Verfügung. Sie erleichtern die Portierung von bereits bestehenden Programmen in eine 16-Bit Windows-Anwendung.

clreol	(conio.h)	*printf*	(stdio.h)
clrscr	(conio.h)	*putch*	(conio.h)
fgetchar	(stdio.h)	*putchar*	(stdio.h)
getch	(stdio.h)	*puts*	(stdio.h)
getchar	(stdio.h)	*scanf*	(stdio.h)
getche	(stdio.h)	*vprintf*	(stdio.h)
gets	(stdio.h)	*vscanf*	(stdio.h)
gotoxy	(conio.h)	*wherex*	(conio.h)
kbhit	(conio.h)	*wherey*	(conio.h)
perror	(errno.h)		

Inline-Routinen

Zu diesen Routinen existieren Inline-Versionen. Der Compiler erzeugt Inline-Code, wenn Sie **#pragma intrinsic** verwenden oder eine Programmoptimierung durchführen. Im Kapitel 2 des *Benutzerhandbuchs* erfahren Sie Einzelheiten zu diesem Thema.

abs	(math.h)	*stpcpy*	(string.h)
alloca	(malloc.h)	*strcat*	(string.h)
_crotl	(stdlib.h)	*strchr*	(string.h)
_crotr	(stdlib.h)	*strcmp*	(string.h)
_lrotl	(stdlib.h)	*strcpy*	(string.h)
_lrotr	(stdlib.h)	*strlen*	(string.h)
memchr	(mem.h)	*strncat*	(string.h)
memcmp	(mem.h)	*strncmp*	(string.h)
memcpy	(mem.h)	*strncpy*	(string.h)
memset	(mem.h)	*strnset*	(string.h)
_rotl	(stdlib.h)	*strrchr*	(string.h)
_rotr	(stdlib.h)	*strset*	(string.h)

Ein-/Ausgabe-Routinen

Diese Routinen stellen Ein-/Ausgabefunktionalität auf Stream- und auf Betriebssystemebene zur Verfügung.

access	(io.h)	*getchar*	(stdio.h)
_rtl_chmod	(io.h)	*getche*	(conio.h)

chmod	(io.h)
chsize	(io.h)
clearerr	(stdio.h)
_rtl_close	(io.h)
close	(io.h)
_rtl_creat	(io.h)
creat	(io.h)
creatnew	(io.h)
creattemp	(io.h)
cscanf	(conio.h)
_dos_close	(dos.h)
_dos_creat	(dos.h)
_dos_creatnew	(dos.h)
_dos_getfileattr	(dos.h)
_dos_getftime	(dos.h)
_dos_open	(dos.h)
_dos_read	(dos.h)
_dos_setfileattr	(dos.h)
_dos_setftime	(dos.h)
_dos_write	(dos.h)
dup	(io.h)
dup2	(io.h)
eof	(io.h)
fclose	(stdio.h)
fcloseall	(stdio.h)
fdopen	(stdio.h)
feof	(stdio.h)
ferror	(stdio.h)
fflush	(stdio.h)
fgetc	(stdio.h)
fgetchar	(stdio.h)
fgetpos	(stdio.h)
fgets	(stdio.h)
filelength	(io.h)
fileno	(stdio.h)
flushall	(stdio.h)
fopen	(stdio.h)
fprintf	(stdio.h)
fputc	(stdio.h)
fputchar	(stdio.h)
fputs	(stdio.h)
fread	(stdio.h)
freopen	(stdio.h)
fscanf	(stdio.h)
fseek	(stdio.h)
fsetpos	(stdio.h)
_fsopen	(stdio.h)
fstat	(sys\stat.h)
ftell	(stdio.h)
fwrite	(stdio.h)
getc	(stdio.h)
getch	(conio.h)
getftime	(io.h)
gets	(stdio.h)
getw	(stdio.h)
ioctl	(io.h)
isatty	(io.h)
kbhit	(conio.h)
lock	(io.h)
locking	(io.h)
lseek	(io.h)
_rtl_open	(io.h)
open	(io.h)
_pclose	(stdio.h)
perror	(stdio.h)
_pipe	(io.h)
_popen	(stdio.h)
printf	(stdio.h)
putc	(stdio.h)
putchar	(stdio.h)
puts	(stdio.h)
putw	(stdio.h)
_rtl_read	(io.h)
read	(io.h)
remove	(stdio.h)
rename	(stdio.h)
rewind	(stdio.h)
rmtmp	(stdio.h)
scanf	(stdio.h)
setbuf	(stdio.h)
setftime	(io.h)
setmode	(io.h)
setvbuf	(stdio.h)
sopen	(io.h)
sprintf	(stdio.h)
sscanf	(stdio.h)
_strerror	(string.h, stdio.h)
strerror	(stdio.h)
tell	(io.h)
tempnam	(stdio.h)
TFile	(file.h)
tmpfile	(stdio.h)
tmpnam	(stdio.h)
umask	(io.h)
unlink	(dos.h)
unlock	(io.h)
utime	(utime.h)
vfprintf	(stdio.h)
vfscanf	(stdio.h)
vprintf	(stdio.h)
vscanf	(stdio.h)
vsprintf	(stdio.h)
vsscanf	(io.h)
_rtl_write	(io.h)

Schnittstellen-routinen

Diese Routinen ermöglichen den Zugriff auf das Betriebssystem, das BIOS und maschinenspezifische Fähigkeiten.

bdos	(dos.h)	*getverify*	(dos.h)
bdosptr	(dos.h)	*inp*	(conio.h)
_bios_equiplist	(bios.h)	*inpw*	(conio.h)
biosequip	(bios.h)	*inport*	(dos.h)
biosmemory	(bios.h)	*inportb*	(dos.h)
_bios_serialcom	(bios.h)	*int86*	(dos.h)
biostime	(bios.h)	*int86x*	(dos.h)
_chain_intr	(dos.h)	*intdos*	(dos.h)
country	(dos.h)	*intdosx*	(dos.h)
ctrlbrk	(dos.h)	*intr*	(dos.h)
_disable	(dos.h)	*MK_FP*	(dos.h)
disable	(dos.h)	*outp*	(conio.h)
dosexterr	(dos.h)	*outpw*	(conio.h)
_dos_getvect	(dos.h)	*outport*	(dos.h)
_dos_setvect	(dos.h)	*outportb*	(dos.h)
_enable	(dos.h)	*parsfnm*	(dos.h)
enable	(dos.h)	*peek*	(dos.h)
FP_OFF	(dos.h)	*peekb*	(dos.h)
FP_SEG	(dos.h)	*poke*	(dos.h)
geninterrupt	(dos.h)	*pokeb*	(dos.h)
getcbrk	(dos.h)	*segread*	(dos.h)
getdfree	(dos.h)	*setcbrk*	(dos.h)
getdta	(dos.h)	*_setcursortype*	(conio.h)
getfat	(dos.h)	*setdta*	(dos.h)
getfatd	(dos.h)	*setvect*	(dos.h)
getpsp	(dos.h)	*setverify*	(dos.h)
getvect	(dos.h)	*sleep*	(dos.h)

Von der locale-Einstellung betroffene API-Routinen

Diese Routinen sind von der aktuellen locale-Einstellung betroffen. Die aktuelle locale-Einstellung wird mit der Funktion *setlocale* festgelegt und mit der Compileroption **-D** __USELOCALES__ eingeschaltet. Wenn Sie __USELOCALES__ definieren, so werden für die folgenden Routinen anstelle von Makros die Funktionsversionen der Laufzeitbibliothek verwendet. Eine Beschreibung der International-API finden Sie im Kapitel 1 des *Programmierhandbuchs*.

cprintf	(stdio.h)	*scanf*	(stdio.h)
cscanf	(stdio.h)	*setlocale*	(locale.h)
fprintf	(stdio.h)	*sprintf*	(stdio.h)
fscanf	(stdio.h)	*sscanf*	(stdio.h)
isalnum	(ctype.h)	*strcoll*	(string.h)
isalpha	(ctype.h)	*strftime*	(time.h)
iscntrl	(ctype.h)	*strlwr*, *_fstrlwr*	(string.h)
isdigit	(ctype.h)	*strupr*, *_fstrupr*	(string.h)
isgraph	(ctype.h)	*strxfrm*	(string.h)
islower	(ctype.h)	*tolower*	(ctype.h)
isprint	(ctype.h)	*toupper*	(ctype.h)

ispunct	(ctype.h)	*vfprintf*	(stdio.h)
isspace	(ctype.h)	*vfscanf*	(stdio.h)
isupper	(ctype.h)	*vprintf*	(stdio.h)
isxdigit	(ctype.h)	*vscanf*	(stdio.h)
localeconv	(locale.h)	*vsprintf*	(stdio.h)
printf	(stdio.h)	*vsscanf*	(stdio.h)

Manipulationsroutinen

Mit diesen Routinen werden Strings und Speicherblöcke kopiert, konvertiert und gesucht.

mblen	(stdlib.h)	*stricmp*	(string.h)
mbstowcs	(stdlib.h)	*string*	(cstring.h)
mbtowc	(stdlib.h)	*strlen*	(string.h)
memccpy	(mem.h, string.h)	*strlwr*	(string.h)
memchr	(mem.h, string.h)	*strncat*	(string.h)
memcmp	(mem.h, string.h)	*strncmp*	(string.h)
memcpy	(mem.h, string.h)	*strncmpi*	(string.h)
memicmp	(mem.h, string.h)	*strncpy*	(string.h)
memmove	(mem.h, string.h)	*strnicmp*	(string.h)
memset	(mem.h, string.h)	*strnset*	(string.h)
movedata	(mem.h, string.h)	*strpbrk*	(string.h)
movmem	(mem.h, string.h)	*strrchr*	(string.h)
setmem	(mem.h)	*strrev*	(string.h)
stpcpy	(string.h)	*strset*	(string.h)
strcat	(string.h)	*strspn*	(string.h)
strchr	(string.h)	*strstr*	(string.h)
strcmp	(string.h)	*strtok*	(string.h)
strcoll	(string.h)	*strupr*	(string.h)
strcpy	(string.h)	*strxfrm*	(string.h)
strcspn	(string.h)	*wcstombs*	(stdlib.h)
strdup	(string.h)	*wctomb*	(stdlib.h)
strerror	(string.h)		

Mathematische Routinen

Die folgenden Routinen führen mathematische Berechnungen und Konvertierungen durch.

abs	(complex.h, stdlib.h)	*labs*	(stdlib.h)
acos	(complex.h, math.h)	*ldexp*	(math.h)
acosl	(math.h)	*ldexpl*	(math.h)
arg	(complex.h)	*ldiv*	(math.h)
asin	(complex.h, math.h)	*log*	(complex.h, math.h)
asinl	(math.h)	*logl*	(math.h)
atan	(complex.h, math.h)	*log10*	(complex.h, math.h)
atan2	(complex.h, math.h)	*log10l*	(math.h)
atan2l	(math.h)	*_lrotl*	(stdlib.h)
atanl	(math.h)	*_lrotr*	(stdlib.h)
atof	(stdlib.h, math.h)	*ltoa*	(stdlib.h)
atoi	(stdlib.h)	*_matherr*	(math.h)
atol	(stdlib.h)	*_matherrl*	(math.h)
_atold	(math.h)	*modf*	(math.h)

bcd	(bcd.h)	*modfl*	(math.h)
cabs	(math.h)	*norm*	(complex.h)
cabsl	(math.h)	*polar*	(complex.h)
ceil	(math.h)	*poly*	(math.h)
ceill	(math.h)	*polyl*	(math.h)
_clear87	(float.h)	*pow*	(complex.h, math.h)
complex	(complex.h)	*pow10*	(math.h)
conj	(complex.h)	*pow10l*	(math.h)
_control87	(float.h)	*powl*	(math.h)
cos	(complex.h, math.h)	*rand*	(stdlib.h)
cosh	(complex.h, math.h)	*random*	(stdlib.h)
coshl	(math.h)	*randomize*	(stdlib.h)
cosl	(math.h)	*real*	(complex.h)
div	(math.h)	*_rotl*	(stdlib.h)
ecvt	(stdlib.h)	*_rotr*	(stdlib.h)
exp	(complex.h, math.h)	*sin*	(complex.h, math.h)
expl	(math.h)	*sinh*	(complex.h, math.h)
fabs	(math.h)	*sinhl*	(math.h)
fabsl	(math.h)	*sinl*	(math.h).h, math.h)
fcvt	(stdlib.h)	*sqrt*	(complex.h, math.h)
floor	(math.h)	*sqrtl*	(math.h)
floorl	(math.h)	*srand*	(stdlib.h)
fmod	(math.h)	*_status87*	(float.h)
fmodl	(math.h)	*strtod*	(stdlib.h)
_fpreset	(float.h)	*strtol*	(stdlib.h)
frexp	(math.h)	*_strtold*	(stdlib.h)
frexpl	(math.h)	*strtoul*	(stdlib.h)
gcvt	(stdlib.h)	*tan*	(complex.h, math.h)
hypot	(math.h)	*tanh*	(complex.h, math.h)
hypotl	(math.h)	*tanhl*	(complex.h, math.h)
imag	(complex.h)	*tanl*	(math.h)
itoa	(stdlib.h)	*ultoa*	(stdlib.h)

Speicherroutinen

Diese Routinen ermöglichen die dynamische Speicherreservierung in den verschiedenen Speichermodellen.

alloca	(malloc.h)	*heapcheckfree*	(alloc.h)
_bios_memsize	(bios.h)	*heapchecknode*	(alloc.h)
calloc	(alloc.h, stdlib.h)	*heapwalk*	(alloc.h)
farcalloc	(alloc.h)	*malloc*	(alloc.h, stdlib.h)
farfree	(alloc.h)	*realloc*	(alloc.h, stdlib.h)
farmalloc	(alloc.h)	*set_new_handler*	(new.h)
free	(alloc.h, stdlib.h)	*stackavail*	(malloc.h)
heapcheck	(alloc.h)		

Verschiedene Routinen

Diese Routinen realisieren nicht-lokale Sprünge sowie die Festlegung der locale-Einstellung.

localeconv	(locale.h)	*setjmp*	(setjmp.h)
longjmp	(setjmp.h)	*setlocale*	(locale.h)

Veraltete Definitionen

Die folgenden globalen Variablen wurden umbenannt, um mit der ANSI-Namensgebung übereinzustimmen. Sie sollten stets die neuen Namen verwenden. Wenn Sie Bibliotheken hinzulinken, die mit Header-Dateien von Borland C++ 3.1 (oder älteren Versionen) compiliert wurden, erhalten Sie die Meldung

```
Error: undefined external varname in module LIBNAME.LIB
```

Ein Bibliotheksmodul, das einen solchen Fehler verursacht, sollte neu compiliert werden. Ist eine Neucompilierung jedoch nicht möglich, so können Sie OBSOLETE.LIB hinzufügen und die externen Variablennamen verfügbar machen.

Die folgenden globalen Variablen wurden umbenannt:

Tabelle 1.4: Veraltete globale Variablen

Alter Name	Neuer Name	Header-Datei
daylight	*_daylight*	*time.h*
directvideo	*_directvideo*	*conio.h*
environ	*_environ*	*stdlib.h*
sys_errlist	*_sys_errlist*	*errno.h*
sys_nerr	*_sys_nerr*	*errno.h*
timezone	*_timezone*	*time.h*
tzname	*_tzname*	*time.h*

Die alten Namen der folgenden Funktionen stehen immer noch zur Verfügung. Der Compiler erzeugt jedoch eine Warnmeldung über die Verwendung eines veralteten Namens. Zukünftige Versionen von Borland C++ unterstützen die alten Funktionsnamen eventuell nicht mehr.

Die folgenden Funktionsnamen wurden geändert:

Tabelle 1.5: Veraltete Funktionsnamen

Alter Name	Neuer Name	Header-Datei
_chmod	*_rtl_chmod*	*io.h*
_close	*_rtl_close*	*io.h*
_creat	*_rtl_creat*	*io.h*
_heapwalk	*_rtl_heapwalk*	*malloc.h*
_open	*_rtl_open*	*io.h*
_read	*_rtl_read*	*io.h*
_write	*_rtl_write*	*io.h*

Routinen zur Prozeßsteuerung

Diese Routinen dienen dazu, neue Prozesse innerhalb von anderen Routinen zu starten und zu beenden.

abort	(process.h)	*exit*	(process.h)
_beginthread	(process.h)	*_expand*	(process.h)
_beginthreadNT	(process.h)	*getpid*	(process.h)

_c_exit	(process.h)	*_pclose*	(stdio.h)
_cexit	(process.h)	*_popen*	(stdio.h)
cwait	(process.h)	*raise*	(signal.h)
_endthread	(process.h)	*signal*	(signal.h)
execl	(process.h)	*spawnl*	(process.h)
execle	(process.h)	*spawnle*	(process.h)
execlp	(process.h)	*spawnlp*	(process.h)
execlpe	(process.h)	*spawnlpe*	(process.h)
execv	(process.h)	*spawnv*	(process.h)
execve	(process.h)	*spawnve*	(process.h)
execvp	(process.h)	*spawnvp*	(process.h)
execvpe	(process.h)	*spawnvpe*	(process.h)
_exit	(process.h)	*wait*	(process.h)

Zeit- und Datumsroutinen

Diese Routinen dienen zur Konvertierung und Manipulation von Zeitangaben.

asctime	(time.h)	*gmtime*	(time.h)
_bios_timeofday	(bios.h)	*localtime*	(time.h)
ctime	(time.h)	*mktime*	(time.h)
difftime	(time.h)	*stime*	(time.h)
_dos_gettime	(dos.h)	*_strdate*	(time.h)
_dos_getdate	(dos.h)	*strftime*	(time.h)
_dos_settime	(dos.h)	*_strtime*	(time.h)
_dos_getdate	(dos.h)	*TDate*	(date.h)
dostounix	(dos.h)	*time*	(time.h)
ftime	(sys\timeb.h)	*TTime*	(time.h)
getdate	(dos.h)	*tzset*	(time.h)
gettime	(dos.h)	*unixtodos*	(dos.h)

variable Argumentlisten

Diese Routinen werden benötigt, um auf Argumentlisten variabler Länge zugreifen zu können (wie bei *printf*, *vprintf*, *vscanf* etc.).

va_arg	(stdarg.h)	*va_start*	(stdarg.h)
va_end	(stdarg.h)		

K A P I T E L 2

Die Funktion main

Im Kapitel 8 des *Programmierhandbuchs* finden Sie Informationen über die Programmierung unter Windows.

Die Funktion *main* muß in jedem C- oder C++ Programm einmal definiert sein. Beim Start rufen Programme mit Textbildschirmausgabe die Funktion *main*, GUI-Programme unter Windows die Funktion *WinMain* auf. Die Position dieser Funktionen innerhalb des Quelltextes spielt eine untergeordnete Rolle. Einige Programmierer plazieren *main* an den Anfang des Quelltextes, andere an das Ende. Unabhängig von der Position der Funktion *main* sind folgende Erläuterungen stets gültig.

Die Parameter von main

main bekommt durch den Startcode des Programms drei Parameter (Argumente) übergeben, die per Konvention als *argc*, *argv* und *env* bezeichnet werden:

- *argc* ist vom Typ **int** und gibt die Anzahl der Kommandozeilen-argumente von *argv* an. Dabei wird der Programmname selbst mitgezählt.
- *argv* ist ein Array aus Stringzeigern (**char***[]):
 - argv[0] enthält den vollen Namen des gestarteten Programms.
 - *argv*[1] zeigt auf den ersten Kommandozeilenparameter.
 - *argv*[2] zeigt auf den zweiten Kommandozeilenparameter.
 - *argv*[*argc-1*] zeigt auf den letzten Parameter, der *main* übergeben wird.
 - *argv*[*argc*] enthält NULL und kennzeichnet so das Ende der Parameterliste.
- *env* ist ebenfalls ein Array aus Stringzeigern. Jedes Element von *env[]* enthält einen Eintrag, der die folgende Form hat: `ENVVAR=`*`Wert`*.

- ENVVAR ist eine Umgebungsvariable, zum Beispiel PATH oder COMSPEC.
- *Wert* ist der Wert, auf den ENVVAR gesetzt ist, zum Beispiel C:\DOS;C:\TOOLS; (für PATH) oder C:\DOS\COMMAND.COM (für COMSPEC).

Die Namen dieser Parameter sind im Prinzip beliebig (obwohl man sich auch hier an existierende Konventionen halten sollte) – ihre Reihenfolge ist dagegen zwingend vorgeschrieben: *argc*, *argv*, *env*.

Die folgenden Beispiele stellen gültige Deklarationen von *main* dar:

```
int main()
int main(int argc)              /* zulässig, aber ungewöhnlich */
int main(int argc, char *argv[])
int main(int argc, char *argv[], char *env[])
```

In den Einträgen zu *getenv* und *putenv* in Kapitel 3 sowie zu *_environ* in Kapitel 4 finden Sie weitere Informationen.

Die Deklaration *int main(int argc)* ist zwar gültig, macht aber nicht viel Sinn – es sei denn, Sie wollen keine Parameter übergeben, sondern nur eine Meldung wie `Kommandozeilenparameter ignoriert` ausgeben.

Der Parameter *env* ist auch über die globale Variable *_environ* verfügbar.

Analoges gilt für *argc* und *argv*: sie sind über die globalen Variablen *_argc* und *_argv* verfügbar.

Die Argumente von main auswerten

Das folgende Beispielprogramm zeigt eine einfache Möglichkeit, wie Sie die der Funktion *main* übergebenen Argumente auswerten können:

```
/* Program ARGS.C */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[])
{
   int i;

   printf("The value of argc is %d \n\n",argc);
   printf("These are the %d command-line arguments passed to"
          "main:\n\n",argc);

   for (i = 0; i < argc; i++)
      printf("   argv[%d]: %s\n", i, argv[i]);

   printf("\nThe environment string(s) on this system are:\n\n");

   for (i = 0; env[i] != NULL; i++)
      printf("   env[%d]: %s\n", i, env[i]);
   return 0;
}
```

Starten Sie dieses Programm wie folgt von der DOS-Kommandozeile aus:

```
C:> args first_arg "arg with blanks" 3  4 "last but one" stop!
```

Leerzeichen werden als Trennzeichen zwischen einzelnen Parametern behandelt – um einen Parameter wie *argv[2]* zu erzeugen, müssen Sie den entsprechenden String in Anführungszeichen einschließen.

Sie erhalten die folgenden Ausgaben (wobei der vollständige Name von ARGS.EXE und die *Environment*-Einträge von Ihrem Computer abhängig sind):

```
The value of argc is 7

These are the 7 command-line arguments passed to main:

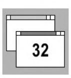
   argv[0]: C:\BC4\ARGS.EXE
   argv[1]: first_arg
   argv[2]: arg with blanks
   argv[3]: 3
   argv[4]: 4
   argv[5]: last but one
   argv[6]: stop!

The environment string(s) on this system are:

   env[0]: COMSPEC=C:\COMMAND.COM
   env[1]: PROMPT=$p $g
   env[2]: PATH=C:\SPRINT;C:\DOS;C:\BC4
```

Die Maximallänge einer Kommandozeile ist auf 255 Zeichen begrenzt, wobei Programmname, Suchwege und Trennzeichen zwischen den einzelnen Parametern mitgezählt werden; diese Begrenzung bezieht sich auf Win32.

Jokerzeichen in Kommandozeilenparametern



Kommandozeilenparameter, die Jokerzeichen enthalten, können auf alle passenden Dateinamen erweitert werden. Dies entspricht in etwa der Arbeitsweise des DOS-Befehls COPY. Sie müssen dazu lediglich die Object-Datei WILDARGS.OBJ zu Ihrem Programm hinzulinken. WILDARGS.OBJ ist Bestandteil von Borland C++.

Argumente mit Jokerzeichen werden nur in Textbildschirm- Anwendungen verwendet.

Wurde WILDARGS.OBJ zu Ihrem Programm hinzugelinkt, so können Sie Ihrer *main*-Funktion Argumente wie etwa *.* übergeben. Das Argument wird dann (im Array *argv*) auf alle Dateinamen erweitert, die der Suchmaske entsprechen. Die Maximalgröße des Array *argv* hängt von der Größe des verfügbaren Heap-Speichers ab.

Wenn keine passenden Dateien gefunden werden, wird das Argument unverändert (inklusive Jokerzeichen) übergeben.

Arumente in Anführungszeichen ("...") werden nicht erweitert.

WILDCARD.OBJ hinzulinken

Die beiden folgenden Befehle compilieren die Datei ARGS.C, linken sie mit WILDARGS.OBJ zusammen und starten anschließend die resultierende Datei ARGS.EXE:

```
BCC ARGS.C WILDARGS.OBJ
ARGS C:\BC4\INCLUDE\*.H "*.C"
```

Beim Aufruf von ARGS.EXE wird das erste Argument zu allen Namen der *.H-Dateien im INCLUDE-Verzeichnis von Borland C++ erweitert. Beachten Sie, daß jeder erweiterte Argumentstring den vollständigen Pfadnamen der jeweiligen Datei enthält. Das Argument *.C wird nicht erweitert, weil es in Anführungszeichen steht.

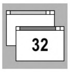
Innerhalb der IDE benötigen Sie für dieses Beispiel eine Projektdatei mit den folgenden beiden Einträgen:

```
ARGS
WILDARGS.OBJ
```

Wenn Sie die Erweiterung der Jokerzeichen per Voreinstellung verwenden wollen, dann müssen Sie Ihre Bibliotheksdateien CW32?.LIB ändern, so daß WILDARGS.OBJ automatisch hinzugelinkt werden. Dazu entfernen Sie SETARGV und INITARGS aus den Bibliotheken und fügen WILDARGS hinzu. Mit den folgenden Befehlen wird TLIB aufgerufen und alle Standard-Bibliotheksdateien werden geändert. Es wird dabei davon ausgegangen, daß sich die Standard-Bibliotheken für C und C++ sowie die Datei WILDARGS.OBJ im aktuellen Verzeichnis befinden.

```
tlib CW32 -setargv +wildargs
tlib CW32MT -setargv +wildargs
```

-p verwenden (Pascal-Aufrufkonventionen)



Beim Compilieren eines Programms, das die Pascal-Aufrufkonventionen verwendet (wie in Kapitel 2 des *Programmierhandbuchs* ausführlich beschrieben), dürfen Sie nicht vergessen, *main* ausdrücklich als C-Routine zu deklarieren. Dies erfolgt mit dem Schlüsselwort **__cdecl:**

```
__cdecl main(int argc, char int * argv[], char int * envp[])
```

Der Rückgabewert von main

Solange ein Programm nicht mit *exit* oder *_exit* abgebrochen wird, ist der Rückgabewert von *main* der Statuscode des Programms und vom Typ **int.** Beim Abbruch eines Programms mit *exit* (oder *_exit*) wird der an *exit* über-

gebene Wert zurückgegeben. Enthält Ihr Programm zum Beispiel den Aufruf

```
exit(1)
```

so ist der Status 1.

Übergabe von Datei-Informationen an Kindprozesse

Wenn Ihr Programm mit den Funktionen *exec* oder *spawn* neue Prozesse erzeugt, so übernehmen diese normalerweise alle offenen Datei-Handles des ursprünglichen Prozesses. Einige Informationen wie der Zugriffsmodus, mit dem diese Dateien geöffnet wurden, gehen jedoch verloren. Wenn Ihr Programm beispielsweise eine Datei binär zum Lesen öffnet und danach mit *spawn* einen Kindprozeß erzeugt, so kann dieser die Datei durch Schreibzugriffe zerstören oder auch im Textmodus aus ihr lesen.

Wollen Sie, daß auch die Kindprozesse solche Informationen über offene Dateien erhalten, so müssen Sie die Datei FILEINFO.OBJ zu Ihrem Programm hinzulinken. Beispielsweise mit:

```
bcc test.c \bc4\lib\fileinfo.obj
```

Die Informationen über offene Dateien werden in der Umgebungsvariablen _C_FILE_INFO übergeben. Diese enthält verschlüsselte Binärinformationen, die Ihr Programm nicht lesen oder ändern sollte. Der Kindprozeß muß mit der C++ Laufzeitbibliothek entwickelt worden sein, um die Information korrekt zu übernehmen. Alle übrigen Programme ignorieren _C_FILE_INFO und erhalten keine Datei-Informationen.

Programme mit mehreren Threads

32-Bit Programme können mehr als einen ausführbaren Thread erzeugen (Multi-Thread-Programme). Wenn Ihr Programm mehrere Threads erzeugt und diese zudem die C++ Laufzeitbibliothek verwenden, so müssen Sie die Bibliothek CW32MT.LIB oder CW32MTI.LIB verwenden.

In der Online-Hilfe erfahren Sie im Beispiel zu *_beginthread*, wie diese Funktionen und *_threadid* verwendet werden.

Die Multi-Thread-Bibliotheken enthalten die Funktionen *_beginthread* und *_beginthreadNT*, die sie zur Erzeugung von Threads benutzen. Die Bibliotheken enthalten zudem die Funktion *_endthread* zum Beenden von Threads sowie die globale Variable *_threadid*, die die Identifikationsnummer des aktuellen Thread enthält (die sogenannte Tread-ID). Die Deklaration von *_threadid* befindet sich in der Header-Datei stddef.h.

Wenn Sie ein Programm mit mehreren Threads compilieren oder linken, dann müssen Sie den Compilerschalter -WM verwenden. Zum Beispiel:

```
bcc32 -WM thread.c
```

Bei der Verwendung der Funktion *signal* in einem Programm mit mehreren Threads müssen Sie vorsichtig sein. In der Beschreibung der Funktion *signal* finden Sie weitere Informationen.

K A P I T E L 3

Die Bibliotheksfunktionen

In der Online-Hilfe finden Sie zu jeder Funktion ein Programmbeispiel, das Sie leicht aus der Hilfe in Ihre Datei kopieren können.

In diesem Kapitel werden die Funktionen der C++ Bibliothek der Borland-Compiler ausführlich beschrieben. Einige Funktionen werden als »Funktionsfamilie« gemeinsam erklärt (z.B. die Funktionen der *exec...*- und *spawn...*-Familie), da sie eine ähnliche Arbeitsweise und Aufgabe haben. Jede andere Routine hat ihren eigenen Eintrag und kann unter ihrem Namen gefunden werden. Werden z.B. die Erläuterungen zur Funktion *free* benötigt, muß in dem entsprechenden Eintrag für *free* nachgeschlagen werden. In diesem Eintrag findet man dann:

- der Name der Include-Datei(en), die den Prototyp von *free* enthalten;
- eine Kurzbeschreibung von *free*;
- die Syntaxdefinition für den Aufruf von *free*;
- eine ausführliche Beschreibung der Arbeitsweise von *free* und der Unterschiede zu anderen Speicherverwaltungsroutinen;
- Informationen zur Portabilität bzw. Verwendung mit anderen Compilern mit entsprechenden Funktionen;
- einen Verweis auf verwandte Funktionen;

Folgendes Beispiel steht stellvertretend für die Struktur eines Funktionseintrages in diesem Handbuch. Die alphabetische Auflistung beginnt auf Seite 45.

Beispielfunktion Header-Dateiname

Hinter dem Funktionsnamen *function* folgt der Name der Header-Datei(en), die den Funktionsprototyp für *function* und eventuell auch Definitionen, Konstanten oder Aufzählungstypen enthält, die von *function* verwendet werden.

Funktion Eine kurze Beschreibung der Funktion.

Syntax `function(modifier parameter[,...]);`

Die hier dargestellte Syntax entspricht dem Prototyp der Funktion. Die Klammer `[,...]` zeigt an, daß noch weitere Parameter folgen können.

Die Portabilität der Funktion wird durch Markierungen (■) in der jeweiligen Spalte einer Portabilitätstabelle angezeigt. Eine Beispieltabelle sieht folgendermaßen aus:

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Jeder Eintrag in der Portabilitätstabelle wird in der folgenden Auflistung beschrieben. Alle weiteren Einschränkungen werden im Abschnitt *Beschreibung* angesprochen.

`DOS`	Auf DOS-Systemen verfügbar.
`UNIX`	Unter UNIX- und/oder POSIX-Systemen verfügbar.
`Win 16`	Kompatibel zu 16-Bit Windows-Programmen, die unter Microsoft Windows 3.1, Windows for Workgroups 3.1 und 3.11 laufen. Wenn Sie mit Easy-Win arbeiten, lesen Sie bitte im *Benutzerhandbuch* nach, wie Sie nicht zu Windows kompatible Funktionen (zum Beispiel *printf* und *scanf*) in Programmen einsetzen können, die unter Windows ausgeführt werden.
`Win 32`	Für 32-Bit Windows Programme unter Win32s 1.0 und Windows NT 3.1 verfügbar.
`ANSI C`	Im ANSI C Standard definiert.
`ANSI C++`	Im ANSI C++ Entwurf enthalten.
`OS/2`	Für OS/2 verfügbar.

Sind in einem Eintrag mehrere Funktionen beschrieben, deren Portabilität aber identisch ist, so wird nur eine Zeile verwendet. Andernfalls wird jede Funktion in einer eigenen Zeile aufgeführt.

Beschreibung Dieser Abschnitt enthält die Aufgabe und die Verwendung der aufgeführten Funktionen, die Beschreibung der benutzten Parameter sowie weitere Details, die Sie für den Einsatz der Funktion und der aufgeführten verwandten Routinen benötigen.

Rückgabewert Typ und Wertebereich des Rückgabewerts (soweit vorhanden) wird hier angegeben. Wenn die Funktion globale Variablen setzt, finden Sie außerdem eine Aufstellung der entsprechenden Werte.

Siehe auch Auflistung anderer Routinen, die mit der beschriebenen Funktion in irgendeinem Zusammenhang stehen. Bei Querverweisen wie *...Funktionsname* oder *Funktionsname...* wird auf ganze Funktionsfamilien hingewiesen (*exec...* bezieht sich also auf alle *exec*-Funktionen wie *execl*, *execle*, *execlp*, *execlpe*, *execv*, *execve*, *execvp* und *execvpe*).

Beispiel Die Funktionsbeispiele finden Sie in der Online-Hilfe. Von dort können Sie sie leicht in Ihre eigenen Anwendungen kopieren.

abort stdlib.h

Funktion Beendet ein Programm mit einer Fehlermeldung.

Syntax `void abort(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *abort* bewirkt einen Programmabbruch durch Aufruf von *raise*(SIGABRT). Falls es für SIGABRT keinen Signal-Handler gibt, so wird eine Abbruchmeldung (`"Abnormal program termination"`) auf *stderr* ausgegeben und daraufhin das Programm durch den Aufruf von *_exit* mit dem Exit-Code 3 beendet.

Rückgabewert Gibt den Exit-Code 3 an den übergeordneten Prozeß oder an den Kommandozeileninterpreter des Betriebssystems zurück.

Siehe auch *assert*, *atexit*, *_exit*, *exit*, *raise*, *signal*, *spawn...*

abs stdlib.h

Funktion Liefert den absoluten Betrag einer Integer-Zahl.

Syntax `int abs(int x);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *abs* liefert den absoluten Betrag seines Integer-Arguments *x*. Wenn stdlib.h aufgenommen wurde, wird diese Funktion als Makro behandelt und somit zu Inline-Code.

Wollen Sie stattdessen die Funktion *abs* verwenden, so nehmen Sie nach `#include<sttlib.h>` die Zeile `#undef abs` in Ihr Programm auf.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert Die Funktion *abs* liefert einen **int**-Wert im Bereich von 0 bis INT_MAX. Ausnahme: Der Wert INT_MIN wird als INT_MIN zurückgegeben. Die Werte für INT_MAX und INT_MIN sind in der Header-Datei limits.h definiert.

Siehe auch *bcd*, *cabs*, *complex*, *fabs*, *labs*

access — io.h

Funktion Ermittelt die Zugriffsmöglichkeiten auf eine Datei.

Syntax

```
int access(const char *filename, int amode);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *access* prüft, ob eine Datei des (als *filename*) angegebenen Namens existiert. Wenn ja, folgt eine Prüfung, ob der in *amode* angegebene Zugriff möglich ist. Die folgenden Werte sind für *amode* definiert:

06 Prüfung auf »Lesen und Schreiben erlaubt«
04 Prüfung auf »Lesen erlaubt«
02 Prüfung auf »Schreiben erlaubt«
01 Prüfung auf »Ausführung erlaubt«
00 Prüfung auf »Datei existiert«

Unter DOS, OS/2 und Windows (16- und 32-Bit) besitzen alle existierenden Dateien automatisch Lesezugriff (*amode* 04), so daß die Werte 04 und 00 für *amode* dasselbe Ergebnis erzeugen. Eine Schreiberlaubnis impliziert bei DOS auch die Erlaubnis zum Lesen – folglich sind die Werte 06 und 02 für *amode* ebenfalls äquivalent.

Wenn *filename* ein Verzeichnis enthält, prüft *access* nur das Vorhandensein dieses Verzeichnisses.

Rückgabewert Ist der durch *amode* angegebene Zugriff erlaubt, so liefert *access* den Wert 0 zurück. Ansonsten ist der Rückgabewert -1, und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

ENOENT Pfad oder Datei nicht vorhanden
EACCES Datei vorhanden, aber Zugriff nicht erlaubt

Siehe auch *chmod*, *fstat*, *stat*

acos, acosl — math.h

Funktion Berechnet den Arcuscosinus.

Syntax

```
double acos(double x);
long double acosl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
acos	■	■	■	■	■	■	■
acosl	■		■	■			■

Beschreibung

acos liefert den Arcuscosinus des Eingabewerts. *acosl* ist die **long double** Version und arbeitet mit einem **long double** Parameter sowie einem **long double** Rückgabewert. Die Argumente zu *acos* und *acosl* müssen sich im Bereich von -1 bis 1 befinden. Andernfalls wird der Wert NAN zurückgeliefert und die globale Variable *errno* erhält den Wert:

EDOM Argument außerhalb des zulässigen Bereichs

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert

Bei einem Argument zwischen -1 und +1 liefern *acos* und *acosl* Werte im Bereich von *0* bis *pi* zurück. Die Fehlerbehandlung für *acos* und *acosl* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

Siehe auch

asin, *atan*, *atan2*, *bcd*, *complex*, *cos*, *_matherr*, *sin*, *tan*

alloca — malloc.h

Funktion

Reserviert temporären Speicher auf dem Stack.

Syntax

```
void *alloca(size_t size);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

alloca reserviert einen Speicherbereich von *size* Bytes auf dem Stack. Der reservierte Speicherbereich wird beim Beenden der aufrufenden Funktion automatisch freigegeben.

alloca modifiziert den Stack-Zeiger und darf deshalb nicht in einem Ausdruck aufgerufen werden, der ein Argument einer Funktion ist.

Die Funktion *alloca* sollte nicht in einem *try*-Block eines C++ Programms eingesetzt werden, da beim Auslösen einer Exception alle von *alloca* auf den Stack gesetzten Werte unbrauchbar werden.

Die aufrufende Funktion muß Referenzen auf lokale Variablen im Stack enthalten, da der Stack andernfalls beim Funktionsende nicht korrekt wiederhergestellt wird, was zu einem Programmabsturz führt. Verwenden Sie deshalb folgenden Programmtext in der aufrufenden Funktion:

```
char *p;
char dummy[1];

dummy[0] = 0;
...
p = alloca(nbytes);
```

Rückgabewert Wenn im Stack ein Speicherbereich von ausreichender Größe zur Verfügung steht, liefert *alloca* einen Zeiger auf den reservierten Speicherbereich zurück. Andernfalls wird der Wert NULL zurückgegeben.

Siehe auch *malloc*

asctime — time.h

Funktion Konvertiert Datum/Uhrzeit in einen String.

Syntax

```
char *asctime(const struct tm *tblock);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *asctime* konvertiert eine Zeitangabe, die als Struktur des Typs *tm* in **tblock* gespeichert ist, in einen String mit 26 Zeichen, der wie *ctime* das folgende Format hat:

```
Wed Jun 23 15:35:54 1993\n\0
```

Rückgabewert *asctime* liefert einen Zeiger auf den String zurück, der Zeit und Datum enthält. Der String ist eine statische Variable, die mit jedem Aufruf von *asctime* überschrieben wird.

Siehe auch *ctime, difftime, ftime, gmtime, localtime, mktime, strftime, stime, time, tzset*

asin, asinl — math.h

Funktion Berechnet den Arcussinus.

Syntax

```
double asin(double x);
long double asinl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
asin	■	■	■	■	■	■	■
asinl	■		■	■			■

Beschreibung *asin* liefert für ein reelles Argument den Arcussinus des eingegebenen Wertes zurück. *asinl* ist die **long double** Version und erwartet ein Argument vom Typ **long double**. *asinl* liefert ein Ergebnis vom Typ **long double** zurück.

asin und *asinl* erwarten ein reelles Argument im Bereich von -1 bis +1. Reelle Argumente außerhalb dieses Bereichs liefern NAN und setzen *errno* auf den Wert

EDOM	Argument außerhalb des zulässigen Bereichs

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert Der Rückgabewert von *asin* und *asinl* liegt für reelle Argumente im Bereich von -*pi*/2 bis *pi*/2. Die Fehlerbehandlung kann für diese Funktionen mit *_matherr* und *_matherrl* modifiziert werden.

Siehe auch *acos*, *atan*, *atan2*, *bcd*, *complex*, *cos*, *_matherr*, *sin*, *tan*

assert — assert.h

Funktion Prüft eine Bedingung und bricht das Programm ab, wenn sie nicht zutrifft.

Syntax

```
void assert(int test);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *assert* ist ein Makro, das bei der Compilierung zu einer **if**Anweisung erweitert wird. Wenn *test* zur Laufzeit des Programms den Wert null (d.h. *false*) ergibt, wird die Meldung

`Assertion failed: `*`test`*`, file `*`Dateiname`*`, line `*`Zeilennummer`*

auf *stderr* ausgegeben. Danach erfolgt der Abbruch des Programms mit *abort*.

In dieser Meldung bezeichnet *Dateiname* den Namen der Quelltextdatei und Zeilennummer die Nummer der Zeile, in der sich der Aufruf von *assert* befindet.

Wenn das Symbol NDEBUG *vor* der Aufnahme von assert.h im Quelltext (oder mit `#define NDEBUG`) definiert ist, entfernt der Präprozessor sämtliche Aufrufe von *assert*.

Rückgabewert Keiner.

Siehe auch *abort*

atan, atanl — math.h

Funktion Berechnet den Arcustangens.

Syntax

```
double atan(double x);
long double atanl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
atan	■	■	■	■	■	■	■
atanl	■		■	■			■

Beschreibung

atan berechnet den Arcustangens des übergebenen Arguments.

atanl ist die **long double** Version und erwartet ein Argument vom Typ **long double**. *atanl* liefert ein Ergebnis vom Typ **long double** zurück. Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert

Der Rückgabewert von *atan* und *atanl* liegt für reelle Argumente im Bereich von *-pi/2* bis *pi/2*. Die Fehlerbehandlung für *atan* und *atanl* kann über die Funktionen *_matherr* und *_matherrl* modifiziert werden.

Siehe auch

acos, asin, atan2, bcd, complex, cos, _matherr, sin, tan

atan2, atan2l — math.h

Funktion

Berechnet den Arcustangens von *y/x*.

Syntax

```
double atan2(double y, double x);
long double atan2l(long double y, long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
atan2	■	■	■	■	■	■	■
atan2l	■		■	■			■

Beschreibung

atan2 liefert den Arcustangens von *y/x* und erzeugt auch dann noch korrekte Resultate, wenn sich der ergebende Winkel *pi/2* oder *-pi/2* nähert (den Polstellen der Tangensfunktion, für Werte von *x* nahe null). Wenn beide Argumente *x* und *y* null sind, setzt die Funktion *errno* auf den Wert EDOM und zeigt so einen ungültigen Eingabewert an.

atan2l ist die **long double** Version und erwartet ein Argument vom Typ **long double**. *atanl* liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert

atan2 und *atan2l* liefern einen Wert im Bereich von *-pi* bis *pi*. Die Fehlerbehandlung kann für diese Funktionen mit *_matherr* und *_matherrl* modifiziert werden.

Siehe auch

acos, asin, atan, cos, _matherr, sin, tan

atexit stdlib.h

Funktion Registriert Routinen, die am Ende eines Programms aufgerufen werden sollen.

Syntax

```
int atexit(void (_USERENTRY * func)(void));
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *atexit* registriert die Funktion, auf die *func* zeigt, als sogenannte Exit-Funktion: bei normalem Programmende ruft *exit* die Funktion (_USERENTRY **func*)() unmittelbar vor der Rückkehr zum Betriebssystem auf. *func* muß mit der _USERENTRY Aufrufkonvention verwendet werden.

Durch jeden Aufruf von *atexit* wird eine weitere Funktion registriert. Bis zu 32 Funktionen können so registriert werden, die bei Programmende nach dem Prinzip »last in – first out« ausgeführt werden, die als letzte mit *atexit* festgelegte Funktion wird also zuerst ausgeführt.

Rückgabewert *atexit* liefert den Wert 0 zurück, wenn die angegebene Funktion aufgenommen werden konnte; falls bereits 32 Funktionen registriert sind (und damit kein weiterer Platz mehr zur Verfügung steht), ist das Ergebnis ungleich null.

Siehe auch *abort*, *_exit*, *exit*, *spawn...*

atof, _atold math.h

Funktion Konvertiert einen String in eine Gleitkommazahl.

Syntax

```
double atof(const char *s);
long double _atold(const char *s);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
atof	■	■	■	■	■	■	■
_atold	■		■	■			■

Beschreibung *atof* konvertiert den durch *s* bezeichneten String in eine Gleitkommazahl und liefert das Ergebnis als **double** zurück. Die Funktion erkennt dabei folgende Elemente in der Zeichendarstellung der Gleitkommazahl:

- eine optionale Folge von Leerzeichen und/oder Tabulatoren;
- ein optionales Vorzeichen;
- eine Folge von Ziffern und einen optionalen Dezimalpunkt (die Ziffern können vor oder hinter dem Dezimalpunkt stehen);

- einen optionalen Exponenten *e* oder *E*, gefolgt von einem optional vorzeichenbehafteten Integerwert.

Der String muß dem folgenden Format entsprechen:

[*whitespace*] [*sign*] [*ddd*] [.] [*ddd*] [*e*|*E*[*sign*]*ddd*]

atof erkennt zusätzlich +*INF* und -*INF* als positive bzw. negative Unendlichkeit sowie +*NAN* und -*NAN* (für »Not a Number« – ein nicht numerisch darstellbarer Wert).

Die Konvertierung bricht bei dem ersten nicht interpretierbaren Zeichen ab.

_atold ist die **long double** Version und konvertiert den durch *s* bezeichneten String in einen Wert vom Typ **long double**.

strtod und *_strtold* arbeiten ähnlich wie *atof* und *_atold*, erlauben aber eine bessere Fehlerbehandlung und werden daher in manchen Anwendungen bevorzugt.

Rückgabewert

atof und *_atold* liefern den konvertierten Wert des Eingabestrings zurück.

Im Fall eines Überlaufs liefert *atof* (oder *_atold*) den Wert +HUGE_VAL oder -HUGE_VAL (bzw. _LHUGE_VAL) zurück, *errno* wird auf ERANGE gesetzt. Die Routine *_matherr* (bzw. *_matherrl*) wird nicht aufgerufen.

Siehe auch

atoi, *atol*, *ecvt*, *fcvt*, *gcvt*, *scanf*, *strtod*

atoi — stdlib.h

Funktion

Konvertiert einen String in eine Integerzahl.

Syntax

```
int atoi(const char *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

atoi konvertiert den durch *s* bezeichneten String in einen **int**. Die Funktion erkennt die folgenden Elemente (in der aufgeführten Reihenfolge):

- optionale führende Leerzeichen und/oder Tabulatoren;
- ein optionales Vorzeichen;
- eine Ziffernfolge.

Der String muß also folgendem Format entsprechen:

[*whitespace*] [*sign*] [*ddd*]

Die Konvertierung endet beim ersten nicht interpretierbaren Zeichen, eine Prüfung auf eventuelle Überläufe findet nicht statt (die Funktionsergebnisse sind dann undefiniert).

Rückgabewert *atoi* liefert das Ergebnis der Konvertierung oder 0, wenn der zu interpretierende String nicht zu einem Integerwert konvertierbar ist.

Siehe auch *atof, atol, ecvt, fcvt, gcvt, scanf, strtod*

atol — stdlib.h

Funktion Konvertiert einen String in einen **long**-Wert.

Syntax

```
long atol(const char *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *atol* konvertiert den durch *s* bezeichneten String in einen **long**-Wert. *atol* erkennt die folgenden Elemente (in der aufgeführten Reihenfolge):

- optionale führende Leerzeichen und/oder Tabulatoren;
- ein (optionales) Vorzeichen;
- eine Ziffernfolge.

Der String muß also folgendem Format entsprechen:

[*whitespace*] [*sign*] [*ddd*]

Die Konvertierung endet beim ersten nicht interpretierbaren Zeichen. Eine Prüfung auf eventuelle Überläufe findet nicht statt (die Funktionsergebnisse sind dann undefiniert).

Rückgabewert *atol* liefert das Ergebnis der Konvertierung zurück. Wenn der zu interpretierende String nicht zu einem **long**-Wert konvertierbar ist, liefert *atol* 0 zurück.

Siehe auch *atof, atoi, ecvt, fcvt, gcvt, scanf, strtod, strtol, strtoul*

_atold

Siehe *atof*.

bdos — dos.h

Funktion Ruft eine DOS-Funktion auf.

Syntax `int bdos(int dosfun, unsigned dosdx, unsigned dosal);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *bdos* ermöglicht direkten Zugriff auf zahlreiche DOS-Systemaufrufe. Spezifische Details zu den einzelnen Funktionen entnehmen Sie bitte dem DOS-Referenzhandbuch.

Für Systemaufrufe, die ein Integerargument erwarten, wird die Funktion *bdos* eingesetzt. Wird ein Zeiger erwartet, benutzen Sie *bdosptr*. In den großen Datenmodellen (Compact, Large) ist es wichtig, *bdosptr* für solche Systemaufrufe einzusetzen, die einen Zeiger als Argument erwarten.

dosfun gibt die Nummer des DOS-Systemaufrufs an und ist in Ihrem DOS-Referenzhandbuch definiert.

dosdx bestimmt den Inhalt des Registers DX.

dosal enthält den Wert des Registers AL.

Rückgabewert Zurückgeliefert wird der Wert, auf den das Register AX von DOS gesetzt wurde.

Siehe auch *bdosptr*, *geninterrupt*, *int86*, *int86x*, *intdos*, *intdosx*

bdosptr — dos.h

Funktion Ruft eine DOS-Funktion auf.

Syntax `int bdosptr(int dosfun, void *argument, unsigned dosal);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *bdosptr* ermöglicht direkten Zugriff auf zahlreiche DOS-Systemaufrufe. Spezifische Details zu den einzelnen Funktionen entnehmen Sie bitte dem DOS-Referenzhandbuch.

Für Systemaufrufe, die ein Integerargument erwarten, wird die Funktion *bdos* eingesetzt. Wird ein Zeiger erwartet, benutzen Sie *bdosptr*. In den großen Datenmodellen (Compact, Large) ist es wichtig, *bdosptr* für solche Systemaufrufe einzusetzen, die einen Zeiger als Argument erwarten. In den kleinen Datenmodellen wird mit *argument* das Register DX gesetzt, in den großen Datenmodellen werden die Werte von DS:DX angegeben, die der Systemaufruf benutzt.

dosfun gibt die Nummer des DOS-Systemaufrufs an und ist in Ihrem DOS-Referenzhandbuch definiert. *dosal* ist der Wert des Registers AL.

Rückgabewert Zurückgeliefert wird der Wert, auf den das Register AX von DOS gesetzt wurde. Tritt bei der Ausführung ein Fehler auf, dann liefert die Funktion den Wert -1, und die globalen Variablen *errno* und *_doserrno* werden entsprechend gesetzt.

Siehe auch *bdos, geninterrupt, int86, int86x, intdos, intdosx*

_beginthread process.h

Funktion Startet die Ausführung eines neuen Thread.

Syntax

```
unsigned long_beginthread(_USERENTRY (*start_address) (void *),
                          unsigned stack_size, void *arglist);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung Die Funktion *_beginthread* erzeugt einen neuen Thread und startet ihn. Die Ausführung des Thread beginnt bei der Adresse *start_address* (Beachten Sie, daß *start_address* als _USERENTRY deklariert sein muß). Der Thread erhält vom Betriebssystem einen Stack der Größe *stack_size* Bytes, wobei der Wert auf das nächste Vielfache von 4096 aufgerundet wird. arglist wird dem Thread als einziger Parameter übergeben. *arglist* kann NULL sein, muß jedoch angegeben werden. Der Thread endet einfach per *return* oder durch den Aufruf von *_endthread*.

➥ Die Funktion *_beginthread* oder *_beginthreadNT* muß anstelle der Thread-erzeugenden API-Funktionen verwendet werden, da nur diese Funktionen die Initialisierungen durchführen, die für die korrekte Arbeitsweise der Funktionen der Laufzeitbibliothek benötigt werden.

Diese Funktion ist nur in den Multi-Thread Bibliotheken enthalten.

Die Funktion ist auch unter OS/2 verfügbar. Dort liefert sie jedoch einen **int**-Wert zurück und benötigt _USERENTRY nicht.

Rückgabewert *_beginthread* liefert die ID des neuen Thread zurück. Im Fehlerfall wird -1 zurückgegeben und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EAGAIN	Zu viele Threads
EINVAL	Ungültige Anforderung

Siehe auch die Win32-Beschreibung von GetLastError.

Siehe auch *_beginthreadNT, _endthread*

_beginthreadNT process.h

Funktion Startet die Ausführung eines neuen Thread unter Windows NT.

Syntax

```
unsigned long _beginthreadNT(void (_USERENTRY *start_address)(void *),
                             unsigned stack_size, void *arglist,
                              void *security_attrib, unsigned long
                               create_flags, unsigned long *thread_id);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			

Beschreibung

Alle Multi-Thread-Programme unter Windows NT müssen *_beginthreadNT* oder *_beginthread* anstelle der Thread-erzeugenden API-Funktionen des Betriebssystems verwenden, da nur diese Funktionen die Initialisierungen durchführen, die für die korrekte Arbeitsweise der Funktionen der Laufzeitbibliothek benötigt werden. Die Funktion *_beginthreadNT* unterstützt zudem die Sicherheitsmechanismen des Betriebssystems.

Die Funktion *_beginthreadNT* erzeugt einen neuen Thread und startet ihn. Die Ausführung des Thread beginnt bei der Adresse *start_address* (Beachten Sie, daß *start_address* als _USERENTRY deklariert sein muß). Der Thread erhält vom Betriebssystem einen Stack der Größe *stack_size* Bytes, wobei der Wert auf das nächste Vielfache von 4096 aufgerundet wurde. *arglist* wird dem Thread als einziger Parameter übergeben. *arglist* kann NULL sein, muß jedoch angegeben werden. Der Thread endet einfach per *return* oder durch den Aufruf von *_endthread*.

Die Funktion verwendet den Zeiger *security_attr* zum Zugriff auf die Struktur SECURITY_ATTRIBUTES. Die Struktur enthält die Sicherheitsattribute des Thread. Hat *security_attr* den Wert NULL, dann wird der Thread mit den Standard-Sicherheitsattributen erzeugt.

Die Funktion wertet den Parameter *create_flags* aus, um zusätzliche Informationen über die Thread-Erzeugung zu erhalten. Ist der Wert dieses Parameters gleich 0, so wird der Thread unmittelbar nach seiner Erzeugung gestartet. Ist *create_flags* gleich CREATE_SUSPENDED, dann wird der Thread erst beim Aufruf der Funktion *ResumeThread* gestartet. *ResumeThread* ist Bestandteil der Win32-API. In der Win32-Beschreibung zu *ResumeThread* finden Sie weitere Informationen.

Die Funktion initialisiert die Variable *thread_id* mit der Thread-ID.

Rückgabewert

_beginthreadNT liefert das Handle des neuen Thread zurück. Im Fehlerfall wird -1 zurückgegeben, und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

B

EAGAIN	Zu viele Threads
EINVAL	Ungültige Anforderung

Siehe auch *_beginthread*, *_endthread*

biosequip — bios.h

Funktion

Liefert eine Liste installierter Peripheriegeräte.

Syntax

```
int biosequip(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Diese Funktion benutzt den BIOS-Interrupt 0x11 und liefert einen Integerwert zurück, der die Systemkonfiguration beschreibt.

Rückgabewert

Der zurückgelieferte Wert setzt sich aus mehreren Bitfeldern zusammen. Die Werte für IBM PC und Kompatible:

Bit 14-15 Anzahl der parallel angeschlossenen Drucker
00 = 0 Drucker
01 = 1 Drucker
10 = 2 Drucker
11 = 3 Drucker

Bit 13 seriell angeschlossener Drucker
Bit 12 Gameport angeschlossen

DOS erkennt normalerweise nur 2 serielle Schnittstellen, kann aber zur Erkennung von 4 Schnittstellen gezwungen werden, der IBM PS/2 kann bis zu 8 Schnittstellen erkennen.

Bit 9-11 Anzahl der seriellen Schnittstellen
000 = 0 Schnittstellen
001 = 1 Schnittstelle
010 = 2 Schnittstellen
011 = 3 Schnittstellen
100 = 4 Schnittstellen
101 = 5 Schnittstellen
110 = 6 Schnittstellen
111 = 7 Schnittstellen

Bit 8 DMA (direct memory access)
0 = Rechner hat DMA
1 = Rechner hat kein DMA, z.B. PC Jr.

Bit 6-7 Anzahl von Laufwerken
00 = 1 Laufwerk
01 = 2 Laufwerke
10 = 3 Laufwerke
11 = 4 Laufwerke, nur falls Bit 0 den Wert 1 hat

Bit 4-5 Bildschirmmodus beim Start
00 = unbenutzt
01 = 40x25 schwarzweiß mit Farbkarte
10 = 80x25 schwarzweiß mit Farbkarte
11 = 80x25 schwarzweiß mit Monochromkarte

Bit 2-3 RAM-Größe auf Hauptplatine
00 = 16 K
01 = 32 K
10 = 48 K
11 = 64 K

Bit 1 Mathematischer Koprozessor installiert
Bit 0 Von Diskette booten

_bios_equiplist — bios.h

Funktion Liefert eine Liste installierter Peripheriegeräte.

Syntax `unsigned _bios_equiplist(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *_bios_equiplist* benutzt den BIOS-Interrupt 0x11 und liefert einen Integerwert zurück, der die Systemkonfiguration beschreibt.

Rückgabewert Der zurückgelieferte Wert setzt sich aus mehreren Bitfeldern zusammen. Die Werte für IBM PC und Kompatible:

Bit 14-15 Anzahl der parallel angeschlossenen Drucker
00 = 0 Drucker
01 = 1 Drucker
10 = 2 Drucker
11 = 3 Drucker

Bit 13 seriell angeschlossener Drucker
Bit 12 Gameport angeschlossen

Bit 9-11 Anzahl der seriellen Schnittstellen
000 = 0 Schnittstellen
001 = 1 Schnittstelle
010 = 2 Schnittstellen
011 = 3 Schnittstellen
100 = 4 Schnittstellen
101 = 5 Schnittstellen

B

	110 = 6 Schnittstellen
	111 = 7 Schnittstellen
Bit 8	DMA (direct memory access)
	0 = Rechner hat DMA
	1 = Rechner hat kein DMA, z.B. PC Jr.
Bit 6-7	Anzahl der Laufwerke
	00 = 1 Laufwerk
	01 = 2 Laufwerke
	10 = 3 Laufwerke
	11 = 4 Laufwerke, nur falls Bit 0 den Wert 1 hat
Bit 4-5	Bildschirmmodus beim Start
	00 = unbenutzt
	01 = 40x25 schwarzweiß mit Farbkarte
	10 = 80x25 schwarzweiß mit Farbkarte
	11 = 80x25 schwarzweiß mit Monochromkarte
Bit 2-3	RAM-Größe auf Hauptplatine
	00 = 16 K
	01 = 32 K
	10 = 48 K
	11 = 64 K
Bit 1	Mathematischer Koprozessor installiert
Bit 0	Von Diskette booten

bioskey — bios.h

Funktion Tastaturabfrage über BIOS-Interrupt.

Syntax

```
int bioskey(int cmd);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *bioskey* verwendet den BIOS-Interrupt 0x16 zur Kommunikation mit der Tastatur. Die Art der auszuführenden Operation wird über den Wert von *cmd* festgelegt.

Rückgabewert Der Rückgabewert hängt von der Operation ab, die über den Wert von *cmd* festgelegt wurde.

0 Falls die niederwertigen 8 Bits ungleich null sind, liefert *bioskey* das nächste ASCII-Zeichen aus dem Tastaturpuffer oder wartet auf die Eingabe eines Zeichens und liefert dieses Zeichen zurück. Bei der Eingabe eines »normalen« Zeichens liefert *bioskey* den entsprechen-

den ASCII-Code im niederwertigen Byte. Für Spezialzeichen wie Funktions- und Cursor-Tasten hat das niederwertige Byte den Wert 0, das höherwertige Byte enthält den erweiterten Tastaturcode (siehe *Technisches Referenzhandbuch* für den IBM PC).

1 prüft, ob ein Zeichen anliegt. Wenn nicht, wird der Wert 0 zurückgeliefert, ansonsten der entsprechende Tastencode. Ein Rückgabewert von 0xFFFFF(-1) bedeutet, daß [*Strg*][*Untbr*] gedrückt wurde. Über diese Funktion gelesene Zeichen werden nicht »abgeholt« – sie verbleiben im Tastaturpuffer und werden beim nächsten Aufruf von bioskey mit dem *cmd*-Wert 0 ausgegeben.

2 liefert den aktuellen Zustand der Umschalttasten ([*Shift*], [*Alt*] usw.). Der Rückgabewert ergibt sich aus der ODER-Verknüpfung folgender Bitwerte:

Bit 7	0x80	[*Einfg*] aktiv
Bit 6	0x40	[*Shift-Lock*] aktiv
Bit 5	0x20	[*NumLock*] aktiv
Bit 4	0x10	[*Rollen*] aktiv
Bit 3	0x08	[*Alt*] gedrückt
Bit 2	0x04	[*Strg*] gedrückt
Bit 1	0x02	[*shift*] links gedrückt
Bit 0	0x01	[*shift*] rechts gedrückt

biosmemory — bios.h

Funktion Liefert die Größe des Hauptspeichers zurück.

Syntax

```
int biosmemory(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung Diese Funktion verwendet den BIOS-Interrupt 0x12 zur Bestimmung der RAM-Größe. Zusätzliches RAM in Grafikkarten, Erweiterungsspeicher und Expansionsspeicher wird nicht berücksichtigt.

Rückgabewert *biosmemory* liefert die Größe des Hauptspeichers in Blöcken von einem KByte zurück.

_bios_memsize — bios.h

Funktion Liefert die Größe des Hauptspeichers zurück.

Syntax

```
unsigned _bios_memsize(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Diese Funktion verwendet den BIOS-Interrupt 0x12 zur Bestimmung der RAM-Größe. Zusätzliches RAM in Grafikkarten, Erweiterungsspeicher und Expansionsspeicher wird nicht berücksichtigt.

Rückgabewert

_bios_memsize liefert die Größe des Hauptspeichers in Blöcken von einem KByte zurück.

biostime — bios.h

Funktion

Liefert den Stand des BIOS-Timers zurück oder setzt einen neuen Wert.

Syntax

```
long biostime(int cmd, long newtime);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Der BIOS-Timer ist ein eingebauter Zähler, der jeweils um 0 Uhr 00 auf den Wert 0 gesetzt und von da an rund 18,2 mal pro Sekunde um eins erhöht wird. *biostime* verwendet den BIOS-Interrupt 0x1A zum Lesen bzw. Setzen dieses Zählers.

Wenn *cmd* den Wert 0 hat, liefert *biostime* den aktuellen Wert des Zählers zurück. Wenn *cmd* den Wert 1 hat, setzt *biostime* den Zähler auf den durch *newtime* angegebenen **long**-Wert.

Rückgabewert

Wenn *cmd* den Wert 0 hat, liefert *biostime* den aktuellen Wert des Zählers zurück.

_bios_timeofday — bios.h

Funktion

Liefert den Stand des BIOS-Timers zurück oder setzt einen neuen Wert.

Syntax

```
unsigned _bios_timeofday(int cmd, long *timep);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Der BIOS-Timer ist ein eingebauter Zähler, der jeweils um 0 Uhr 00 auf den Wert 0 gesetzt und von da an rund 18,2 mal pro Sekunde um eins erhöht wird. *_bios_timeofday* verwendet den BIOS-Interrupt 0x1A zum Lesen bzw. Setzen dieses Zählers.

Der Parameter *cmd* kann einen der folgenden Werte haben:

_TIME_GETCLOCK	*_bios_timeofday* speichert den aktuellen Stand des Zählers an der Position, auf die *timep* zeigt. Wurde der Zähler seit 0 Uhr 00 nicht gesetzt oder gelesen, liefert *_bios_timeofday* 1 zurück, andernfalls ist das Ergebnis 0.
_TIME_SETCLOCK	*_bios_timeofday* setzt den Zähler auf den durch *timep* bezeichneten **long**-Wert. Die Funktion liefert keinen Wert zurück.

Rückgabewert

Die Funktion *_bios_timeofday* liefert den Wert zurück, auf den das Register AX durch die Abfrage des BIOS-Timers gesetzt wurde.

bsearch — stdlib.h

Funktion

Binäres Absuchen eines Array.

Syntax

```
void *bsearch(const void *key, const void *base, size_t nelem,size_t width,
              int (_USERENTRY *fcmp)(const void *, const void *));
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

bsearch sucht eine Tabelle (ein Array) mit *nelem* Elementen ab. Die Elemente müssen sortiert sein. *bsearch* liefert die Adresse des ersten Eintrags in der Tabelle, der mit dem Suchschlüssel **key* übereinstimmt oder 0, wenn keine Übereinstimmung gefunden wird. Da es sich um eine binäre Suche handelt, ist die erste Fundstelle nicht notwendigerweise der erste Eintrag in der Tabelle.

Der Datentyp *size_t* ist in der Header-Datei stddef.h definiert.

Der Parameter *base* muß auf das erste Element des Array zeigen, *nelem* ist ein Integer, der die Anzahl der Array-Elemente enthält. *width* gibt die Größe eines einzelnen Elements in Bytes an.

Die Vergleichsroutine *fcmp* muß mit der _USERENTRY Aufrufkonvention verwendet werden.

Der Parameter **fcmp* zeigt auf eine vom Programmierer erstellte Routine, der jeweils zwei Argumente in Form von Zeigern (*elem1*, *elem2*) zum Vergleich übergeben werden: Die Routine vergleicht **elem1* und **elem2* und liefert für *bsearch* einen Integerwert mit folgenden möglichen Werten zurück:

- < 0 falls **elem1* < **elem2*
- == 0 falls **elem1* == **elem2*
- > 0 falls **elem1* > **elem2*

Rückgabewert *bsearch* liefert die Adresse des ersten Elements des Array zurück, das mit dem Suchschlüssel übereinstimmt.

Falls der gesuchte Eintrag nicht existiert, wird 0 zurückgeliefert.

Siehe auch *lfind*, *lsearch*, *qsort*

C

cabs, cabsl — math.h

Funktion Berechnet den Absolutwert einer komplexen Zahl.

Syntax

```
double cabs(struct complex z);
long double cabsl(struct _complexl z);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
cabs	■	■	■	■			■
cabsl	■		■	■			■

Beschreibung *cabs* ist ein Makro, das den Absolutwert von *z*, einer Struktur des Typs *complex*, berechnet. Dieser Strukturtyp ist in math.h folgendermaßen definiert:

```
struct complex {
   double x, y;
};

struct _complexl {
   long double x, y;
};
```

wobei *x* der reelle und *y* der imaginäre Bestandteil von *z* ist.
Der Aufruf von *cabs* entspricht einem Aufruf von *sqrt* mit den Quadraten der reellen und imaginären Bestandteile von z, also

```
sqrt(z.x * z.x + z.y * z.y)
```

cabsl ist die **long double** Version und erwartet als Argument eine Struktur des Typs *_complexl*. Sie liefert ein Ergebnis vom Typ **long double** zurück.

➥ Wenn Sie mit C++ arbeiten, können Sie auch den Typ *complex* (definiert in complex.h) und die Funktion *abs* zur Ermittlung des Absolutwerts einer komplexen Zahl verwenden.

Rückgabewert *cabs* (oder *cabsl*) liefert den Absolutwert von *z* als **double**. Bei einem Überlauf liefert *cabs* (oder *cabsl*) HUGE_VAL (bzw. _LHUGE_VAL) und setzt *errno* auf

ERANGE Ergebnis außerhalb des Rechenbereichs

Die Fehlerbehandlung für diese Funktionen kann über *_matherr* und *_matherrl* modifiziert werden.

Siehe auch *abs*, *complex*, *errno* (globale Variable), *fabs*, *labs*, *_matherr*

calloc — stdlib.h

Funktion Reserviert Hauptspeicher.

Syntax

```
void *calloc(size_t nitems, size_t size);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Über diese Funktion wird der *Heap* verwaltet, in dem Platz für dynamische Variablen reserviert und wieder freigegeben werden kann. Viele Datenstrukturen, wie z.B. Bäume und Listen, können nur auf dem Heap untergebracht werden.

Speichermodelle sind nur in 16-Bit Anwendungen verfügbar.

In den kleinen Datenmodellen (Small, Medium) liegt der Heap zwischen dem Ende des Datensegments und der Spitze des Stack. Der Heap belegt den gesamten hier zur Verfügung stehenden Speicherbereich bis auf einige Bytes direkt unterhalb der Stack-Spitze, die für Stack-Vergrößerungen und für das Betriebssystem reserviert sind.

In den großen Datenmodellen (Compact, Large) beginnt der Heap oberhalb des Stack und endet mit der höchsten vom System zur Verfügung gestellten Speicheradresse.

calloc reserviert einen Speicherbereich von *nitems * size* Bytes, die mit 0 initialisiert werden. Für die Reservierung von Speicherblöcken mit mehr als 64 KByte muß die Funktion *farcalloc* verwendet werden.

Rückgabewert *calloc* liefert einen Zeiger auf den neu reservierten Speicherbereich. Wenn kein Speicherbereich ausreichender Größe zur Verfügung steht, oder wenn *nitems* oder *size* den Wert 0 haben, liefert *calloc* NULL zurück.

Siehe auch *farcalloc*, *free*, *malloc*, *realloc*

ceil, ceill — math.h

Funktion Rundet auf.

Syntax

```
double ceil(double x);
long double ceil(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
ceil	■	■	■	■	■	■	■
ceill	■		■	■			■

Beschreibung *ceil* ermittelt den kleinsten Integerwert größer oder gleich *x*. *ceill* ist die **long double** Version und erwartet ein Argument vom Typ **long double**. Sie liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert Die Funktionen *ceil* und *ceill* liefern ihr ganzzahliges Ergebnis als Wert vom Typ **double** bzw. **long double** zurück.

Siehe auch *floor*, *fmod*

_c_exit process.h

Funktion Führt die gleichen Arbeiten wie *_exit* durch, ohne das laufende Programm zu beenden.

Syntax

```
void _c_exit(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_c_exit* führt die gleichen Arbeiten wie *_exit* durch, beendet aber den aufrufenden Prozeß nicht.

Rückgabewert Keiner.

Siehe auch *abort*, *atexit*, *_cexit*, *exec...*, *exit*, *_exit*, *signal*, *spawn...*

_cexit process.h

Funktion Führt die gleichen Arbeiten wie *exit* durch, ohne das laufende Programm zu beenden.

Syntax

```
void _cexit(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_cexit* führt die gleichen Arbeiten wie *exit* durch, schließt aber keine offenen Dateien und beendet den aufrufenden Prozeß nicht. Sämtliche Datei-

puffer werden geschrieben, und die mit *atexit* festgelegten Exit-Funktionen werden aufgerufen.

Rückgabewert Keiner.

Siehe auch *abort*, *atexit*, *_c_exit*, *exec...*, *_exit*, *exit*, *signal*, *spawn...*

cgets — conio.h

Funktion Liest einen String von der Tastatur.

Syntax

```
char *cgets(char *str);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *cgets* liest einen String von der Tastatur und speichert ihn (und die Stringlänge) an der Speicherstelle, auf die *str* zeigt.

cgets liest die Zeichen, bis es eine CR/LF-Kombination antrifft oder bis die maximal zulässige Zeichenzahl gelesen wurde. Vor der Speicherung des Strings wird die eventuell gelesene CR/LF-Kombination durch ein abschließendes Nullzeichen (\0) ersetzt.

Speichern Sie vor dem Aufruf von *cgets* in *str[0]* die Maximallänge des zu lesenden Strings. Nach der Rückkehr steht in *str[1]* die Anzahl der tatsächlich gelesenen Zeichen. Die gelesenen Zeichen werden ab *str[2]* gespeichert und enden mit einem Nullzeichen. *str* muß daher mindestens *str[0]* + 2 Bytes lang sein.

Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert *cgets* liefert einen Zeiger auf *str[2]* zurück, d.h. auf die Adresse des ersten eingelesenen Zeichens.

Siehe auch *cputs*, *fgets*, *getch*, *getche*, *gets*

_chain_intr — dos.h

Funktion Übergibt die Steuerung an einen anderen Interrupt-Handler.

Syntax

```
void _chain_intr(void(interrupt far *newhandler)());
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *_chain_intr* übergibt die Steuerung vom aktuell ausgeführten Interrupt-Handler an den Interrupt-Handler mit der Adresse *newhandler*. Der aktuelle Registersatz wird dabei *nicht* an den neuen Handler übergeben. Der neue Interrupt-Handler übernimmt stattdessen die Register, die vom alten Handler auf dem Stack plaziert und dort möglicherweise modifiziert wurden. Der neue Interrupt-Handler kann zurückkehren, als wäre er der alte Handler. Dieser wird nicht wieder aktiviert.

_chain_intr kann nur von C-Interrupt-Funktionen aufgerufen werden. Die Funktion ist hilfreich beim Schreiben eines speicherresidenten Programms (TSR), das sich selbst in die Kette der Interrupt-Handler (z.B. dem Tastatur-Interrupt) einhängen muß.

Rückgabewert *_chain_intr* liefert keinen Wert zurück.

Siehe auch *_dos_getvect*, *_dos_setvect*

chdir — dir.h

Funktion Wechselt das aktuelle Verzeichnis.

Syntax

```
int chdir(const char *path);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *chdir* setzt das durch *path* bezeichnete Verzeichnis als aktuelles Arbeitsverzeichnis. Der Parameter *path* muß ein existierendes Verzeichnis bezeichnen.

path kann auch eine Laufwerksangabe enthalten, wie etwa:

```
chdir ("a:\\BC")
```

Hier wird jedoch nur das aktuelle Verzeichnis dieses Laufwerks gewechselt, nicht das aktive Laufwerk selbst.

Die Funktion ändert nur das aktuelle Verzeichnis des Elternprozesses.

Rückgabewert Wenn das angegebene Verzeichnis gesetzt werden konnte, liefert *chdir* den Wert 0; ansonsten wird -1 zurückgeliefert, und die globale Variable *errno* erhält den Wert:

ENOENT Pfad/Datei nicht gefunden

Siehe auch *getcurdir*, *getcdw*, *getdisk*, *mkdir*, *rmdir*, *setdisk*, *system*

_chdrive — direct.h

Funktion Setzt das aktuelle Laufwerk.

Syntax

```
int _chdrive(int drive);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_chdrive* macht das durch *drive* bezeichnete Laufwerk (1 = A, 2 = B, 3 = C, usw.) zum aktuellen Laufwerk.

Diese Funktion ändert nur das aktuelle Laufwerk des Elternprozesses.

Rückgabewert Wenn das aktuelle Laufwerk erfolgreich geändert werden konnte, liefert *_chdrive* den Wert 0; ansonsten wird -1 zurückgeliefert.

Siehe auch *_dos_setdrive*

_chmod — dos.h, io.h

Veraltete Funktion. Siehe *_rtl_chmod*.

chmod — sys\stat.h

Funktion Ändert die Zugriffsmöglichkeiten auf eine Datei.

Syntax

```
int chmod(const char *path, int amode);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *chmod* setzt die Zugriffsberechtigung der durch *path* angegebenen Datei auf den durch *amode* angegebenen Wert. *path* zeigt auf einen String.

amode kann eine (oder beide) der in sys\stat.h definierten symbolischen Konstanten S_IWRITE und S_IREAD enthalten.

Wert von *amode*	Zugriffsberechtigung
S_IWRITE	Schreiben erlaubt
S_IREAD	Lesen erlaubt
S_IREAD \| S_IWRITE	Lesen und Schreiben erlaubt

Schreiberlaubnis beinhaltet Leseerlaubnis.

Rückgabewert Bei fehlerfreier Ausführung liefert *chmod* den Wert 0. Im Fehlerfall ist der Rückgabewert -1, und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht erlaubt
ENOENT	Pfad/Datei nicht gefunden

Siehe auch *access, fstat, open, sopen, _rtl_chmod, stat*

C

chsize — io.h

Funktion Ändert die Größe einer Datei.

Syntax

```
int chsize(int handle, long size);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *chsize* ändert die Größe der mit *handle* verbundenen Datei. Wenn diese Datei mehr als *size* Bytes umfaßt, wird sie entsprechend verkürzt (d.h. alle Zeichen hinter dem neuen Dateiende-Flag werden gelöscht). Andernfalls wird sie mit Nullzeichen ('\0') aufgefüllt.

Die Datei muß in beiden Fällen in einem Modus geöffnet worden sein, der Schreibaktionen erlaubt.

Rückgabewert Bei fehlerfreier Ausführung liefert *chsize* den Wert 0. Ansonsten ist der Rückgabewert -1 und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht erlaubt
EBADF	Ungültiges Handle
ENOSPC	Nicht genügend Platz vorhanden

Siehe auch *close, creat, open, _rtl_creat*

_clear87 — float.h

Funktion Löscht das Statuswort für Gleitkommabearbeitung.

Syntax

```
unsigned int _clear87(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_clear87* löscht das Statuswort für Gleitkommabearbeitung, dessen Inhalt sich aus dem Statuswort des 80x87 und einigen Bits zusammensetzt, die von der Routine zur Behandlung von Ausnahmezuständen des 80x87 entdeckt werden.

Rückgabewert Es wird der vorherige Inhalt des Statuswortes zurückgeliefert, dessen einzelne Bits in float.h dokumentiert sind.

Siehe auch *_control87*, *_fpreset*, *_status87*

clearerr stdio.h

Funktion Setzt das Fehler- und das Dateiende-Flag für einen bestimmten Stream zurück.

Syntax `void clearerr(FILE *stream);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *clearerr* setzt das Fehler-Flag und das Dateiende-Flag des durch *stream* bezeichneten Stream auf 0 zurück. Das Fehler-Flag einer Datei bleibt nach einem Fehler solange gesetzt, bis entweder *clearerr* oder *rewind* aufgerufen werden. Das Dateiende-Flag wird bei jeder Eingabeoperation in die Datei zurückgesetzt.

Rückgabewert Keiner.

Siehe auch *eof*, *feof*, *ferror*, *perror*, *rewind*

clock time.h

Funktion Berechnet die seit dem Programmaufruf vergangene »CPU-Zeit«.

Syntax `clock_t clock(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *clock* kann verwendet werden, um die zwischen zwei Ereignissen vergangene Zeit zu bestimmen. Um die Zeit in Sekunden zu berechnen, kann die von *clock* gelieferte Zeit durch die Konstante CLK_TCK geteilt werden.

Rückgabewert *clock* liefert die seit dem Programmaufruf bis jetzt vergangene »CPU-Zeit« zurück. Falls die Prozessorzeit nicht verfügbar ist oder deren Wert nicht dargestellt werden kann, liefert die Funktion -1 zurück.

Siehe auch *time*

_close io.h

Veraltete Funktion. Siehe *_rtl_close*.

close io.h

Funktion Schließt eine Datei.

Syntax
```
int close(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *close* schließt die durch *handle* bezeichnete Datei. Ein Handle wird durch einen Aufruf von *_rtl_creat*, *creat*, *creatnew*, *creattemp*, *dup*, *dup2*, *open* oder *_rtl_open* einer Datei zugeordnet.

Diese Funktion schreibt kein ^Z (*Ctrl-Z*) als Dateiende-Zeichen. Falls Sie eine Datei mit ^Z abschließen wollen, müssen Sie dieses Zeichen explizit ausgeben.

Rückgabewert *close* liefert den Wert 0 zurück, wenn die Operation fehlerfrei ausgeführt werden konnte, ansonsten -1.

handle muß ein gültiges Handle sein, dem eine gültige, offene Datei zugeordnet ist. Ist dies nicht der Fall, so erhält die globale Variable *errno* den Wert:

EBADF Ungültiges Handle

Siehe auch *chzize*, *creat*, *creatnew*, *dup*, *fclose*, *open*, *_rtl_creat*, *sopen*

closedir dirent.h

Funktion Schließt einen Verzeichnis-Stream.

Syntax
```
int closedir(DIR *dirp);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung Unter UNIX ist *closedir* auf POSIX-Systemen verfügbar. Die Funktion schließt den Verzeichnis-Stream *dirp*, der vorher durch einen Aufruf von

opendir geöffnet wurde. Nachdem der Stream geschlossen wurde, zeigt *dirp* nicht mehr auf einen gültigen Verzeichnis-Stream.

Rückgabewert Bei fehlerfreier Ausführung liefert *closedir* den Wert 0. Ansonsten ist der Rückgabewert -1, und die globale Variable *errno* wird auf den folgenden Wert gesetzt:

EBADF	Der Parameter *dirp* zeigt nicht auf einen gültigen geöffneten Verzeichnis-Stream.

Siehe auch *errno* (globale Variable), *opendir*, *readdir*, *rewinddir*

clreol conio.h

Funktion Löscht alle Zeichen von der aktuellen Cursor-Position bis zum Zeilenende.

Syntax

```
void clreol(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *clreol* löscht innerhalb des aktuellen Textfensters die Spalten von der Cursor-Position bis zum Ende der Zeile. Der Cursor wird dabei nicht bewegt.

Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *clrscr*, *delline*, *window*

clrscr conio.h

Funktion Löscht den Inhalt des aktuellen Textfensters.

Syntax

```
void clrscr(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *clrscr* löscht den gesamten Inhalt des aktuellen Textfensters und plaziert den Cursor in die obere linke Ecke des Textfensters, d.h. auf die Koordinaten (1,1).

Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *clreol*, *delline*, *window*

_control87 float.h

Funktion Liest und setzt das Steuerwort für Gleitkommaoperationen.

Syntax

```
unsigned int _control87(unsigned int newcw,
                        unsigned int mask);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_control87* ermittelt oder ändert das Steuerwort für Gleitkommaoperationen.

Über die Bits dieses Worts – ein **unsigned int**-Wert – werden insbesondere die Modi für Genauigkeit, Rundungsrichtung und die Behandlung unendlicher Werte in den Gleitkommaroutinen festgelegt. Durch Änderung dieser Modi können Ausnahmesituationen bei Gleitkommarechnungen unterschiedlich behandelt werden.

Die einzelnen Bits von *mask* bestimmen, welche Bits des Steuerwortes verändert werden. Die entsprechenden Bits von *newcw* definieren die neuen Werte. Wenn ein Bit von *mask* auf 1 gesetzt ist, dann wird das entsprechende Bit des Steuerwortes auf den Wert gesetzt, den das korrespondierende Bit von *newcw* hat. Ein Beispiel:

Ursprüngliches Steuerwort:	0100	0011	0110	0011
mask:	1000	0001	0100	1111
newcw:	1110	1001	0000	0101
Geänderte Bits	1*xxx*	*xxx*1	*x*0*xx*	0101

Wenn *mask* 0 ist, liefert *_control87* das Steuerwort zurück, ohne es zu ändern.

Rückgabewert *_control87* liefert das neue Steuerwort zurück, das als Bitmuster interpretiert werden muß. Eine vollständige Definition aller zurückgelieferten Bits finden Sie in der Header-Datei float.h.

Siehe auch *_clear87*, *_fpreset*, *signal*, *_status87*

cos, cosl math.h

Funktion Berechnet den Cosinus des Arguments.

Syntax

```
double cos(double x);
long double cosl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
cos	■	■	■	■	■	■	■
cosl	■		■	■			■

Beschreibung

cos erwartet einen Winkel in der Einheit *rad* als Argument und liefert den dazugehörigen Cosinus zurück.

cosl ist die **long double** Version und erwartet ein Argument vom Typ **long double**. Sie liefert ein Ergebnis vom Typ **long double** zurück.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert

Für ein reelles Argument liefert *cos* Werte im Bereich von -1 bis +1 zurück. Die Fehlerbehandlung für *cos* und *cosl* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

Siehe auch

acos, asin, atan, atan2, bcd, complex, _matherr, sin, tan

cosh, coshl — math.h

Funktion

Berechnet den Cosinus hyperbolicus des Arguments.

Syntax

```
double cosh(double x);
long double coshl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
cosh	■	■	■	■	■	■	■
coshl	■		■	■			■

Beschreibung

cosh berechnet den Cosinus hyperbolicus $(e^{x} + e^{-x}) / 2$. *coshl* ist die **long double** Version und erwartet ein Argument vom Typ **long double**. Sie liefert ein Ergebnis vom Typ **long double** zurück.

Diese Funktion kann mit den Datentypen *bed* und *complex* verwendet werden.

Rückgabewert

Bei fehlerfreier Ausführung liefert *cosh* den Cosinus hyperbolicus des Arguments.

Bei Überläufen liefert *cosh* den Wert HUGE_VAL und *coshl* den Wert _LHUGE_VAL (jeweils mit korrektem Vorzeichen) zurück, und die globale Variable *errno* wird auf ERANGE (Ergebnis außerhalb des Rechenbereichs) gesetzt. Die Fehlerbehandlung für *cosh* und *coshl* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

Siehe auch *acos, asin, atan, atan2, bcd, complex, cos, _matherr, sin, sinh, tan, tanh*

country dos.h

Funktion

Liefert Informationen über nationale Formate für Uhrzeit, Datum und numerische Werte.

Syntax

```
struct COUNTRY *country(int xcode, struct COUNTRY *cp);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung

Die Funktion *country* ist unabhänig von der aktuellen locale Einstellung (Funktion *setlocale*).

country setzt nationale Formate für numerische Werte, Trennzeichen, Uhrzeit und Datum, wobei die tatsächlich gesetzten Werte von der verwendeten Betriebssystemversion abhängig sind.

Wenn *cp* den Wert -1 hat, wird das landesspezifische Format auf den durch *xcode* angegebenen Wert gesetzt, der in diesem Fall ungleich null sein muß. Ansonsten füllt *country* die Struktur, auf die *cp* zeigt: Wenn *xcode* den Wert 0 hat, entsprechen die Werte den aktuellen landesspezifischen Einstellungen, wenn *xcode* ungleich null ist, werden die Werte des entsprechenden Landes ermittelt.

Die Struktur COUNTRY ist in dos.h wie folgt definiert:

```
struct COUNTRY {
   int  co_date;        /* Datumsformat */
   char co_curr[5];     /* Währungssymbol */
   char co_thsep[2];    /* Trennzeichen für 1000er */
   char co_desep[2];    /* Dezimaltrennzeichen */
   char co_dtsep[2];    /* Datumstrennzeichen */
   char co_tmsep[2];    /* Uhrzeittrennzeichen */
   char co_currstyle;   /* Währungsformat */
   char co_digits;      /* Anzahl Nachkommastellen bei Währung */
   char co_time;        /* Uhrzeit: 12 / 24 Stunden */
   long co_case;        /* Groß-/Kleinschreibung */
   char co_dasep[2];    /* Datentrennzeichen */
   char co_fill[10];    /* Füllzeichen */
};
```

Der Integer *co_date* kann drei Werte annehmen:

0 USA-Format: Monat, Tag, Jahr

1 europäisches Format: Tag, Monat, Jahr

2 japanisches Format: Jahr, Monat, Tag

Das Element *co_currstyle* legt fest, in welchem Format Geldbeträge ausgegeben werden:

0 das Währungssymbol steht ohne Leerzeichen vor dem Betrag ("DM1000");
1 das Währungssymbol steht ohne Leerzeichen hinter dem Betrag ("1000DM");
2 wie 0, nur mit einem Leerzeichen zwischen Währungssymbol und Betrag;
3 wie 1, nur mit einem Leerzeichen zwischen Betrag und Währungssymbol.

Rückgabewert Bei fehlerfreier Ausführung liefert *country* das Zeigerargument *cp* zurück, andernfalls NULL.

cprintf conio.h

Funktion Gibt formatiert auf den Bildschirm aus.

Syntax

```
int cprintf(const char *format[, argument, ...]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *cprintf* übernimmt eine Reihe von Argumenten, auf die jeweils eine Formatierungsanweisung angewendet wird. Die Formatierungsanweisungen sind in dem String enthalten, auf den *format* zeigt. Die so formatierten Daten werden direkt im aktuellen Textfenster ausgegeben. Die Anzahl der übergebenen Argumente muß mit der der Formatierungsanweisungen übereinstimmen.

Weitere Details über Formatierungsanweisungen finden Sie unter *printf*.

Abhängig vom Wert der globalen Variablen *_directvideo* wird der String entweder direkt in den Bildschirmspeicher geschrieben oder über den Umweg eines BIOS-Aufrufs ausgegeben.

Anders als bei *fprintf* und *printf* wird keine Übersetzung des Zeichens *n* (Zeilenvorschub) in *r**n* (Wagenrücklauf/Zeilenvorschub) vorgenommen. Ebenso wird das Zeichen *t* (Tabulator) nicht in Leerzeichen umgewandelt.

➡ Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert *cprintf* liefert die Anzahl der ausgegebenen Zeichen zurück.

Siehe auch *_directvideo* (globale Variable), *fprintf*, *printf*, *putch*, *sprintf*, *vprintf*

cputs conio.h

Funktion Gibt einen String auf den Bildschirm aus.

Syntax `int cputs(const char *str);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

cputs schreibt den nullterminierten String *str* auf den Bildschirm in das aktuelle Textfenster. Diese Funktion erzeugt *keinen* automatischen Zeilenvorschub am Ende der Ausgabe.

Abhängig vom Wert der globalen Variablen *_directvideo* wird der String entweder direkt in den Bildschirmspeicher geschrieben oder über den Umweg eines BIOS-Aufrufs ausgegeben. Anders als bei *puts* wird keine Übersetzung des Zeichens *\n* (Zeilenvorschub) in *\r\n* (Wagenrücklauf/Zeilenvorschub) vorgenommen.

➥ Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert *cputs* liefert das zuletzt geschriebene Zeichen zurück.

Siehe auch *cgets*, *_directvideo* (globale Variable), *fputs*, *putch*, *puts*

_creat — io.h

Veraltete Funktion. Siehe *_rtl_creat*.

creat — io.h

Funktion Erzeugt eine neue Datei oder überschreibt eine bereits vorhandene Datei.

Syntax `int creat(const char *path, int amode);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

creat erzeugt eine neue Datei oder leitet das Überschreiben einer existierenden Datei ein, die durch *path* bezeichnet wird. Der Parameter *amode* wird nur bei neu erzeugten Dateien ausgewertet.

Mit *creat* erzeugte Dateien werden grundsätzlich in dem durch die globale Variable *_fmode* gesetzten Modus (O_TEXT oder O_BINARY) erzeugt.

Existiert eine Datei des angegebenen Namens noch nicht, so wird sie von *creat* erzeugt. Wenn die Datei bereits existiert und das Schreiben-Attribut gesetzt ist, wird sie auf null Bytes verkürzt, und die Dateiattribute blei-

ben unverändert. Falls bei der Datei das Nur-Lesen-Attribut gesetzt ist, bricht *creat* mit einem Fehler ab, die Datei bleibt unverändert.

Ein Aufruf von *creat* überprüft nur das S_IWRITE-Bit von *amode*. Ist dieses Bit 1, kann in die Datei geschrieben werden, ist das Bit 0, ist nur das Lesen erlaubt. Alle anderen Dateiattribute des Betriebssystems werden auf 0 gesetzt.

Die folgenden Konstanten sind in sys\stat.h für *amode* definiert:

Wert von *amode*	Zugriffsberechtigung
S_IWRITE	Schreiben erlaubt
S_IREAD	Lesen erlaubt
S_IREAD \| S_IWRITE	Lesen und Schreiben erlaubt

Die Erlaubnis zum Schreiben enthält automatisch die Leseerlaubnis.

Rückgabewert

Wenn die angegebene Datei fehlerfrei geöffnet werden konnte, liefert *creat* das neue Datei-Handle (einen positiven Integerwert) zurück.

Im Fehlerfall wird -1 geliefert und *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht erlaubt
ENOENT	Pfad/Datei nicht gefunden
EMFILE	Zu viele Dateien bereits offen

Siehe auch

chmod, *chsize*, *close*, *_rtl_creat*, *creatnew*, *creattemp*, *dup*, *dup2*, *_fmode* (globale Variable), *fopen*, *open*, *sopen*, *write*

creatnew — io.h

Funktion

Erzeugt eine neue Datei.

Syntax

```
int creatnew(const char *path, int mode);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

creatnew arbeitet wie *_rtl_creat* – mit einer Ausnahme: Wenn die angegebene Datei bereits existiert, wird mit einer Fehlermeldung abgebrochen, die Datei bleibt unverändert.

Der Parameter *mode* kann gleich null oder eine ODER-Kombination der folgenden Konstanten sein, die in dos.h definiert sind:

FA_HIDDEN	Verborgene Datei
FA_RDONLY	Nur-Lesen-Attribut
FA_SYSTEM	Systemdatei

Rückgabewert

Wenn die angegebene Datei fehlerfrei geöffnet werden konnte, liefert *creatnew* das neue Datei-Handle (einen positiven Integerwert) zurück.

Im Fehlerfall wird -1 zurückgeliefert und die globale Variable *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht erlaubt
EEXIST	Datei existiert bereits
EMFILE	Zu viele Dateien bereits offen
ENOENT	Pfad/Datei nicht gefunden

Siehe auch

close, *_rtl_creat*, *creat*, *creattemp*, *_dos_creatnew, dup*, *_fmode* (globale Variable), *open*

creattemp io.h

Funktion

Erzeugt eine neue Datei im angegebenen Verzeichnis, wobei die Namensvergabe automatisch erfolgt.

Syntax

```
int creattemp(char *path, int attrib);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

Mit *creattemp* erzeugte Dateien werden grundsätzlich in dem durch die globale Variable *_fmode* gesetzten Modus (O_TEXT oder O_BINARY) erzeugt.

Denken Sie daran, daß ein umgekehrter Schrägstrich als '\\' im Argument path erscheinen muß.

Das als *path* übergebene Argument bezeichnet ein Verzeichnis, endet mit einem umgekehrten Schrägstrich und sollte lang genug für den resultierenden Dateinamen sein. Durch *creattemp* erzeugte Dateien werden nach dem Ablauf des Programms *nicht* automatisch gelöscht.

creattemp durchsucht das angegebene Verzeichnis und erzeugt eine Datei, deren Name innerhalb dieses Verzeichnisses eindeutig ist, *path* wird danach auf den Namen dieser Datei gesetzt.

Der Funktion *creattemp* kann mit *attrib* ein Wort mit DOS-Dateiattributen übergeben werden. Nach einer fehlerfreien Erzeugung zeigt der Dateizeiger auf den Anfang der Datei. Die Datei wird für Lese- und Schreibzugriffe geöffnet.

Der Parameter *attrib* kann gleich null oder eine ODER-Kombination der folgenden Konstanten sein, die in dos.h definiert sind:

FA_HIDDEN	Verborgene Datei
FA_RDONLY	Nur-Lesen-Attribut
FA_SYSTEM	Systemdatei

Rückgabewert Wenn die angegebene Datei fehlerfrei erstellt werden konnte, liefert *creattemp* das neue Datei-Handle (einen positiven Integerwert) zurück.

Im Fehlerfall wird -1 zurückgeliefert und die globale Variable *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht erlaubt
ENOENT	Pfad/Datei nicht gefunden
EMFILE	Zu viele Dateien bereits offen

Siehe auch *close*, *_rtl_creat*, *creat*, *creatnew*, *dup*, *_fmode* (globale Variable), *open*

_crotl, _crotr — stdlib.h

Funktion Rotiert einen **unsigned char**-Wert nach links oder rechts.

Syntax

```
unsigned char _crotl(unsigned char val, int count);
unsigned char _crotr(unsigned char val, int count);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Funktionen *_crotl* und *_crotr* rotieren den übergebenen Wert *val* um *count* Bits nach links bzw. rechts.

Das Argument *val* ist ein **unsigned char**-Wert, oder seine dezimale oder hexadezimale Entsprechung.

Rückgabewert Die Funktion liefert den rotierten Wert zurück:

- *_crotl* liefert den um *count* Bits nach links rotierten Wert von *val* zurück.
- *_crotl* liefert den um *count* Bits nach rechts rotierten Wert von *val* zurück.

Siehe auch *_lrotl, _lrotr, _rotl, _rotr*

cscanf — conio.h

Funktion Liest und formatiert Eingaben von der Tastatur.

Syntax

```
int cscanf(char *format[, address, ...]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

cscanf liest mehrere Eingabefelder zeichenweise direkt von der Tastatur ein. Jedes Feld wird dabei entsprechend einer Formatierungsanweisung formatiert, die *cscanf* im Formatstring *format* übergeben wird. Die formatierten Eingaben werden an Adressen, die nach dem Parameter *format* angegeben werden, gespeichert und direkt auf dem Bildschirm ausgegeben. Die Anzahl der übergebenen Adressen und der Formatierungsanweisungen muß übereinstimmen.

Details zu Formatierungsanweisungen finden Sie unter scanf.

cscanf kann das Lesen eines bestimmten Eingabefeldes abbrechen, bevor das normale Feldende (ein Whitespace-Zeichen) erreicht ist. *cscanf* kann auch aus verschiedenen Gründen den Lesevorgang vollständig abbrechen. Die möglichen Ursachen hierfür werden unter *scanf* erläutert.

Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert

cscanf liefert die Anzahl der fehlerfrei gelesenen und gespeicherten Felder zurück. Gelesene, aber nicht gespeicherte Felder werden nicht mitgezählt. Wurden überhaupt keine Eingabefelder gespeichert, so wird der Wert 0 zurückgegeben.

Der Versuch, über das Ende einer Datei hinauszulesen, liefert den Wert EOF.

Siehe auch

fscanf, *getche*, *scanf*, *sscanf*

ctime — time.h

Funktion

Konvertiert Datum und Uhrzeit in einen String.

Syntax

```
char *ctime(const time_t *time);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

ctime erwartet einen Zeiger auf Datum und Uhrzeit des Systems (das ist der Wert, der von der Funktion *time* geliefert wird). Das Ergebnis von *ctime* ist ein String mit 26 Zeichen, der mit einem Zeilenvorschub und einem Nullzeichen abgeschlossen ist:

```
Sat Jul 03 11:31:54 1993\n\0
```

Alle Felder dieses Strings haben eine konstante Länge.

Die globale **long**-Variable *_timezone* enthält den Unterschied zwischen der lokalen Zeit und der *Greenwich Mean Time* (GMT) in Sekunden (für MEZ ist *_timezone* -1*60*60). Die globale Variable *_daylight* darf nur dann auf einen Wert ungleich null gesetzt werden, wenn die Standardkonvertierung der USA für Sommer- und Winterzeit angewendet werden soll. Diese Variablen werden nicht von einem Benutzerprogramm direkt, sondern von der Funktion *tzset* gesetzt

Rückgabewert *ctime* liefert einen Zeiger auf einen statischen String zurück, der Datum und Uhrzeit enthält und mit jedem Aufruf dieser Funktion erneut gesetzt wird.

Siehe auch *asctime*, *_daylight* (globale Variable), *difftime*, *ftime*, *getdate*, *gmtime*, *localtime*, *settime*, *time*, *_timezone* (globale Variable), *tzset*

ctrlbrk — dos.h

Funktion Setzt die Behandlungsroutine für [*Strg*][*Untbr*].

Syntax

```
void ctrlbrk(int (*handler)(void));
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *ctrlbrk* definiert die Funktion, auf die *handler* zeigt, als neue Behandlungsroutine von [*Strg*][*Untbr*]. Der Interrupt-Vektor 0x23 wird so modifiziert, daß er die angegebene Funktion aufruft.

ctrlbrk richtet einen neuen DOS-Interrupt-Handler ein, der die angegebene Funktion aufruft; diese Funktion wird nur indirekt aufgerufen.

Über *ctrlbrk* gesetzte C-Routinen können beliebige Schritte und Systemaufrufe enthalten, anstelle eines Rücksprungs ist auch die Verzweigung zu anderen Programmteilen (mit *longjmp*) möglich. Die durch `[Strg][Untbr]` aufgerufene Routine kann entweder 0 (Abbruch des Programms) oder einen Wert ungleich 0 (Fortsetzung des Programms) zurückliefern.

Rückgabewert Keiner.

Siehe auch *getcbrk*, *signal*

cwait — process.h

Funktion Wartet auf das Ende eines Kindprozesses.

Syntax

```
int cwait(int *statloc, int pid, int action);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung

Die Funktion *cwait* wartet auf das Ende des Kindprozesses mit der Prozeß-ID *pid*. An der Adresse *statloc* wird (sofern nicht NULL) der Status des beendeten Prozesses gespeichert. Über den Parameter *action* wird festgelegt, ob nur auf das Ende des angegebenen Prozesses oder auch auf das seiner Kinder gewartet wird.

Wird der Kindprozeß normal beendet (durch Aufruf von *exit* oder Rücksprung aus *main*), so ist das Statuswort wie folgt definiert:

Bit 0-7 Null

Bit 8-15 Das niederwertige Byte des Rückgabewerts des Kindprozesses. Das ist der durch *exit* oder den Rücksprung aus *main* erhaltene Wert. Wird der Kindprozeß durch einfachen Rücksprung aus *main* (ohne Wert) verlassen, so ist der Inhalt dieser Bits unbestimmt.

Wird der Kindprozeß nicht normal beendet, dann ist das Statuswort wie folgt definiert:

Bits 0-7 Information über die Beendigung des Kindprozesses

1 Abbruch durch kritischen Fehler

2 Ausführungfehler, Schutzverletzung

3 Externes Signal

Bits 8-15 Null

Ist *pid* gleich null, so wartet *cwait* auf das Ende aller Kindprozesse. Andernfalls enthält *pid* die Prozeß-ID des Kindes, auf das gewartet wird. Diesen Wert muß man durch einen vorherigen Aufruf einer asynchronen *spawn*-Funktion ermittelt haben.

Gültige Werte für *action* sind WAIT_CHILD (nur auf den angegebenen Kindpro- zeß warten) und WAIT_GRANDCHILD (auf den angegebenen Kindprozeß *und* alle seine Kinder warten). Beide Werte sind in der Header-Datei process.h definiert.

Rückgabewert

Bei einer normalen Beendigung des Kindprozesses liefert *cwait* die Prozeß-ID des Kindes zurück.

Bei einer anomalen Beendigung des Kindprozesses liefert die Funktion -1 an den Elternprozeß zurück und setzt *errno* auf EINTR (anomales Ende des Kindprozesses).

Wenn *cwait* zurückkehrt, ohne daß ein Kindprozeß beendet wurde, so liefert die Funktion -1 zurück und setzt *errno* auf einen der folgenden Werte:

ECHILD	Kein Kind vorhanden oder *pid* ungültig
EINVAL	Wert für *action* ist ungültig

Siehe auch *spawn*, *wait*

delline — conio.h

Funktion Entfernt eine Zeile im Textfenster.

Syntax

```
void delline(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *delline* löscht die Zeile, in der sich der Cursor befindet, und verschiebt danach alle Zeilen unter der gelöschten um eine Position nach oben. *delline* arbeitet im aktuell aktiven Textfenster.

➡ Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *clreol*, *clrscr*, *insline*, *window*

difftime — time.h

Funktion Berechnet die Differenz zwischen zwei Zeitangaben.

Syntax

```
double difftime(time_t time2, time_t time1);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *difftime* berechnet die zwischen *time1* und *time2* liegende Zeit in Sekunden.

Rückgabewert *difftime* liefert die Differenz zweier Zeiten als **double**-Wert zurück.

Siehe auch *asctime*, *ctime*, *_daylight* (globale Variable), *gmtime*, *localtime*, *time*, *_timezone* (globale Variable)

disable, _disable, enable, _enable — dos.h

D

Funktion Deaktiviert und aktiviert Hardware-Interrupts.

Syntax

```
void disable(void);
void _disable(void);
void enable(void);
void _enable(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung Diese Makros stellen dem Programmierer eine flexible Steuerung der Hardware-Interrupts zur Verfügung.

Die Makros *disable* und *_disable* deaktivieren alle Hardware-Interrupts. Lediglich NMIs (nicht maskierbare Interrupts) von Peripheriegeräten werden noch bearbeitet.

enable und *_enable* aktivieren alle Hardware-Interrupts.

Rückgabewert Keiner.

Siehe auch *getvect*

div — stdlib.h

Funktion Dividiert zwei ganzzahlige Werte und liefert den Quotienten und den Divisionsrest zurück.

Syntax

```
div_t div(int numer, int denom);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *div* dividiert die als *numer* (Dividend) und *denom* (Divisor) übergebenen Integerwerte und liefert Quotienten und Divisionsrest in einer Struktur des Typs *div_t* zurück. Dieser Datentyp ist in stdlib.h folgendermaßen definiert:

```
typedef struct {
   int quot;      /* Quotient (Divisionsergebnis) */
   int rem;       /* Divisionsrest */
} div_t;
```

Rückgabewert *div* liefert eine Struktur mit den Elementen *quot* (Quotient) und *rem* (Divisionsrest) zurück.

Siehe auch *ldiv*

_dos_close dos.h

Funktion Schließt eine Datei.

Syntax

```
unsigned _dos_close(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_close* schließt die durch *handle* bezeichnete Datei. *handle* ist ein Datei-Handle, das durch einen Aufruf von *_dos_creat*, *_dos_creatnew* oder *_dos_open* einer Datei zugeordnet wurde.

Rückgabewert *_dos_close* liefert den Wert 0 zurück, wenn die Operation fehlerfrei ausgeführt werden konnte. Andernfalls liefert die Funktion den Fehlercode des Betriebssystems zurück, und die globale Variable *errno* erhält den Wert:

EBADF Ungültiges Handle

Siehe auch *_dos_creat*, *_dos_open*, *_dos_read*, *_dos_write*

_dos_commit dos.h

Funktion Schreibt gepufferte Ausgaben einer Datei auf die Diskette/Festplatte.

Syntax

```
unsigned dos_commit(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *_dos_commit* veranlaßt DOS, alle zu einem Datei-Handle gehörenden gepufferten Ausgaben physikalisch auf die Diskette/Festplatte zu schreiben.

Rückgabewert *_dos_commit* liefert bei erfolgreicher Beendigung den Wert 0 zurück. Im Fehlerfall wird der DOS-Fehlercode zurückgeliefert und die globale Variable *errno* wird auf EBADF gesetzt.

Siehe auch *_dos_creat*, *_dos_write*, *_rtl_close*, *_rtl_creat*,

_dos_creat dos.h, io.h

Funktion Erzeugt eine neue Datei oder überschreibt eine bereits vorhandene Datei.

Syntax

```
unsigned _dos_creat(const char *path, int attrib, int *handlep);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung Die Funktion *_dos_creat* öffnet die durch *path* bezeichnete Datei grundsätzlich im binären Modus. Nach einer fehlerfreien Öffnung bzw. Erzeugung zeigt der Dateizeiger auf das erste Byte der Datei. *_dos_creat* speichert das Datei-Handle an der durch *handlep* bezeichneten Position. Die Datei wird für Lese- und Schreibzugriffe geöffnet.

Wenn die Datei bereits existiert, wird sie auf null Bytes Länge verkürzt (d.h. der vorherige Inhalt wird gelöscht).

Der Parameter *attrib* enthält eine ODER-Kombination der folgenden in dos.h definierten Konstanten:

_A_NORMAL	Normale Datei
_A_RDONLY	Nur-Lesen-Datei
_A_HIDDEN	Verborgene Datei
_A_SYSTEM	Systemdatei

Rückgabewert *_dos_creat* liefert bei fehlerfreier Ausführung 0, andernfalls wird der Fehlercode des Betriebssystems zurückgeliefert.

Im Fehlerfall wird die globale Variable errno auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht erlaubt
ENOENT	Pfad/Datei nicht gefunden
EMFILE	Zu viele offene Dateien

Siehe auch *chsize*, *close*, *creat*, *creatnew*, *creattemp*, *_rtl_chmod*, *_rtl_close*

_dos_creatnew — dos.h

Funktion Erzeugt eine neue Datei.

Syntax

```
unsigned _dos_creatnew(const char *path, int attrib, int *handlep);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_creatnew* erzeugt und öffnet die durch *path* bezeichnete Datei im binären Modus. Die Datei erhält die Zugriffsberechtigung *attrib* (ein Attributwort des Betriebssystems). Nach einer fehlerfreien Erzeugung wird das Datei-Handle an der durch *handlep* bezeichneten Adresse gespeichert, und der Dateizeiger zeigt auf den Anfang der Datei. Die Datei wird für Lese- und Schreibzugriffe geöffnet.

Existiert die Datei bereits, so liefert *_dos_creatnew* einen Fehler zurück, und die Datei bleibt unverändert.

Der Parameter *attrib* enthält eine ODER-Kombination aus einer oder mehreren der folgenden in dos.h definierten Konstanten:

_A_NORMAL	Normale Datei
_A_RDONLY	Nur-Lesen-Datei
_A_HIDDEN	Verborgene Datei
_A_SYSTEM	Systemdatei

Rückgabewert Wenn die angegebene Datei fehlerfrei geöffnet werden konnte, liefert *_dos_creatnew* 0 zurück. Andernfalls wird der Fehlercode des Betriebssystems zurückgeliefert, und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht erlaubt
EEXIST	Datei existiert bereits
ENOENT	Pfad/Datei nicht gefunden
EMFILE	Zu viele Dateien bereits offen

Siehe auch *creatnew*, *_dos_close*, *_dos_creat*, *_dos_getfileattr*, *_dos_setfileattr*

dosexterr — dos.h

Funktion Liefert einen erweiterten Fehlercode von DOS zurück.

Syntax

```
int dosexterr(struct DOSERROR *eblkp);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *dosexterr* speichert in der Struktur DOSERROR nach einer fehlerhaften DOS-Operation zusätzliche Informationen. Die Struktur ist folgendermaßen definiert:

```
struct DOSERROR {
   int  de_exterror;   /* erweiterter Fehlercode */
   char de_class;      /* Fehlerklasse */
   char de_action;     /* Aktion */
   char de_locus;      /* Fehlerstelle */
};
```

Die in dieser Struktur gespeicherten Werte werden durch die DOS-Funktion 0x59 ermittelt. Ein Wert von 0 für *de_exterror* zeigt an, daß die zuletzt ausgeführte DOS-Operation fehlerfrei verlaufen ist.

Rückgabewert *dosexterr* liefert den Wert von *de_exterror* zurück.

_dos_findfirst dos.h

D

Funktion Durchsucht ein Dateiverzeichnis.

Definition

```
unsigned _dos_findfirst(const char *pathname, int attrib,
                        struct fint_t *ffblk);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_findfirst* startet die Suche nach einer Datei in einem Verzeichnis.

Der Parameter *pathname* ist ein String und bezeichnet den zu suchenden Eintrag. Er kann optional eine Laufwerksbezeichnung und/oder einen Pfad enthalten, Jokerzeichen (* und ?) sind erlaubt. Falls eine Datei gefunden wird, werden in der über *ffblk* angegebenen Struktur *find_t* die im Verzeichnis enthaltenen Dateiinformationen gespeichert.

Die Struktur *find_t* ist folgendermaßen definiert:

```
struct find_t {
    char reserved[21];     /* vom Betriebssystem verwendet */
    char attrib;           /* Attribut */
    int wr_time;           /* Zeit der letzten Änderung */
    int wr_date;           /* Datum der letzten Änderung */
    long size;             /* Dateigröße in Bytes */
    char name[13];         /* Name der gefundenen Datei */
};
```

Der Parameter *attrib* enthält das Datei-Attributwort des Betriebssystems und wird dazu verwendet, bestimmte Dateien bei der Suche auszuwählen. *attrib* enthält eine ODER-Kombination der folgenden in dos.h definierten Konstanten:

_A_NORMAL	Normale Datei
_A_RDONLY	Nur-Lesen-Attribut
_A_HIDDEN	Verborgene Datei
_A_SYSTEM	Systemdatei
_A_VOLID	Name des Datenträgers
_A_SUBDIR	Verzeichnis
_A_ARCH	Archiv-Bit

Ausführliche Informationen über diese Attribute finden Sie in den Handbüchern Ihres Betriebssystems.

Beachten Sie, daß *wr_time* und *wr_date* Bitfelder für den Zeit- bzw. Datumseintrag der Datei enthalten. Die Struktur dieser Bitfelder ist durch das Betriebssystem vorgegeben.

wr_time:

Bits 0 bis 4	Sekunden geteilt durch 2 (d.h. 10 bedeutet hier 20 Sekunden)
Bits 5 bis 10	Minuten
Bits 11 bis 15	Stunden

wr_date:

Bits 0 bis 4	Tag
Bits 5 bis 8	Monat
Bits 9 bzw. 15	Jahre seit 1980 (d.h. 9 bedeutet hier 1989)

Rückgabewert *_dos_findfirst* liefert den Wert 0 zurück, wenn ein zu *pathname* passender Eintrag gefunden wurde. Wenn kein Eintrag gefunden wird oder der angegebene Dateiname aus irgendeinem Grund ungültig ist, wird der Fehlercode des Betriebssystems als Ergebnis geliefert, und *errno* erhält den Wert:

ENOENT Pfad/Datei nicht gefunden

Siehe auch *_dos_findnext*

_dos_findnext — dos.h

Funktion Setzt eine *_dos_findfirst*-Suche fort.

Definition

```
unsigned _dos_findnext(struct find_t *ffblk);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung Mit *_dos_findnext* können Verzeichnisse nach weiteren Einträgen abgesucht werden, die zum Parameter *pathname* einer vorangegangenen *_dos_findfirst*-Suche passen. *ffblk* zeigt auf die gleiche Struktur wie bei *_dos_findfirst* und enthält die notwendigen Informationen zur Fortsetzung der Suche. *_dos_findnext* liefert bei jedem Aufruf einen weiteren Dateieintrag zurück – solange, bis das Verzeichnis keine weiteren Einträge mehr enthält, die dem Parameter *pathname* von *_dos_findfirst* entsprechen.

Rückgabewert *_dos_findnext* liefert den Wert 0 zurück, wenn ein weiterer, zu *pathname* passender Eintrag gefunden wurde. Sind keine weiteren Einträge vorhanden, oder ist der angegebene Dateiname aus irgendeinem Grund ungültig, so wird der Fehlercode des Betriebssystems zurückgeliefert und *errno* erhält den Wert:

ENOENT Datei/Pfad nicht gefunden

Siehe auch *_dos_findfirst*

_dos_getdate, _dos_setdate, getdate, setdate — dos.h

Funktion Ermittelt oder setzt das Systemdatum.

Definition

```
void _dos_getdate(struct dosdate_t *datep);
unsigned _dos_setdate(struct dosdate_t *datep);
void getdate(struct date *datep);
void setdate(struct date *datep);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

D

Beschreibung *getdate* füllt die durch *datep* angegebene *date*-Struktur mit dem aktuellen Systemdatum.

setdate setzt das Systemdatum (Monat, Tag und Jahr) auf durch *datep* angegebenen Einträge einer *date*-Struktur.

Die Struktur *date* ist folgendermaßen definiert:

```
struct date {
   int  da_year;  /* aktuelles Jahr */
   char da_day;   /* Tag des Monats */
   char da_mon;   /* Monat (1 = Januar) */
};
```

_dos_getdate füllt durch *datep* angegebene *dosdate_t*-Struktur mit dem aktuellen Systemdatum.

Die Struktur *dosdate_t* ist folgendermaßen definiert:

```
struct dosdate_t {
   unsigned char day;       /* 1 - 31 */
   unsigned char month;     /* 1 - 12 */
   unsigned int  year;      /* 1980 - 2099 */
   unsigned char dayofweek; /* 0 - 6 (0 = Sonntag) */
};
```

Rückgabewert *_dos_getdate*, *getdate* und *setdate* liefern keinen Wert zurück.

_dos_setdate liefert 0 zurück, wenn das Datum erfolgreich gesetzt werden konnte. Andernfalls wird ein Wert ungleich null zurückgeliefert und die globale Variable *errno* erhält den Wert:

EINVAL Ungültiges Datum

Siehe auch *ctime*, *gettime*, *settime*

_dos_getdiskfree dos.h

Funktion Ermittelt den freien Platz auf einer Diskette/Festplatte.

Syntax

```
unsigned _dos_getdiskfree(unsigned char drive, struct diskfree_t *dtable);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_getdiskfree* ermittelt Informationen über den freien Platz auf dem durch *drive* angegebenen Laufwerk und speichert sie in der durch *dtable* bezeichneten Struktur des Typs *diskfree_t*. Wenn *drive* den Wert 0 hat, wird das aktuelle Laufwerk angesprochen, ansonsten gilt A = 1, B = 2 usw.

Die Struktur *diskfree_t* ist folgendermaßen definiert:

```
struct diskfree_t {
   unsigned avail_clusters;      /* freie Cluster */
   unsigned total_clusters;      /* Gesamtzahl Cluster */
   unsigned bytes_per_sector;    /* Bytes pro Sektor */
   unsigned sectors_per_cluster; /* Sektoren pro Cluster */
};
```

Rückgabewert *_dos_getdiskfree* liefert bei fehlerfreier Ausführung 0 zurück. Andernfalls wird ein Wert ungleich null zurückgeliefert und *errno* erhält den Wert:

EINVAL Ungültiges Laufwerk

Siehe auch *getfat*, *getfatd*

_dos_getdrive, _dos_setdrive dos.h

Funktion Liest bzw. ändert die aktuelle Laufwerksnummer.

Syntax

```
void _dos_getdrive(unsigned *drivep);
void _dos_setdrive(unsigned drivep, unsigned *ndrives);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_getdrive* ermittelt die aktuelle Laufwerksnummer.

_dos_setdrive setzt die aktuelle Laufwerksnummer und speichert die Gesamtzahl der Laufwerke an der Adresse, auf die *ndrives* zeigt.

Für die Laufwerksnummern in der durch *drivep* angegebenen Adresse gilt: 1 = A, 2 = B, 3 = C, usw.

_dos_setdrive ändert das aktuelle Laufwerk des Elternprozesses.

Rückgabewert Keiner. Mit *_dos_getdrive* können Sie feststellen, ob das aktuelle Laufwerk erfolgreich geändert wurde.

Siehe auch *getcwd*

_dos_getfileattr, _dos_setfileattr — dos.h

D

Funktion Ändert den Zugriffsmodus einer Datei.

Syntax

```
int _dos_getfileattr(const char *path, unsigned *attribp);
int _dos_setfileattr(const char path, unsigned attrib);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_getfileattr* liest die Dateiattribute der durch *path* angegebenen Datei. Die Attribute werden an der durch *attribp* angegebenen Adresse gespeichert.

_dos_setfileattr setzt die Zugriffsberechtigung der durch *path* angegebenen Datei auf den Wert von *attrib*. Die Dateiattribute bestehen aus einer ODER-Kombination der folgenden in dos.h definierten symbolischen Konstanten:

_A_RDONLY	Nur-Lesen-Attribut
_A_HIDDEN	Verborgene Datei
_A_SYSTEM	Systemdatei
_A_VOLID	Datenträgername
_A_SUBDIR	Verzeichnis
_A_ARCH	Archiv-Bit
_A_NORMAL	Normale Datei (keine Attribut-Bits gesetzt)

Rückgabewert Bei fehlerfreier Ausführung liefern *_dos_getfileattr* und *_dos_setfileattr* als Ergebnis 0. Andernfalls wird der Fehlercode des Betriebssystems zurückgeliefert und die globale Variable *errno* erhält den Wert:

ENOENT	Pfad/Datei nicht gefunden

Siehe auch *chmod, stat*

_dos_getftime, _dos_setftime — dos.h

Funktion Liest bzw. setzt Datum und Uhrzeit einer Datei.

Syntax

```
unsigned _dos_getftime(int handle, unsigned *datep, unsigned *timep);
unsigned _dos_setftime(int handle, unsigned date, unsigned time);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung

_dos_getftime liest das Datum und die Uhrzeit der durch *handle* angegebenen Datei. Diese Datei muß vorher mit *_dos_open*, *_dos_creat* oder *_dos_creatnew* geöffnet worden sein. *_dos_getftime* speichert das Datum und die Zeit an den durch *datep* und *timep* angegebenen Adressen.

_dos_setftime setzt Datum und Uhrzeit der Datei auf die neuen, in den Parametern *time* und *date* angegebenen Werte.

Beachten Sie, daß die Datums- und Uhrzeitwerte Bitfelder für die Uhrzeit und das Datum der Datei enthalten. Die Struktur dieser Bitfelder ist durch das Betriebssystem vorgegeben.

Datum:

Bits 0 bis 4	Tag
Bits 5 bis 8	Monat
Bits 9 bis 15	Jahre seit 1980 (d.h. 9 bedeutet hier 1989)

Uhrzeit:

Bits 0 bis 4	Resultierende Sekunden geteilt durch 2 (d.h. 10 bedeutet hier 20 Sekunden)
Bits 5 bis 10	Minuten
Bits 11 bis 15	Stunden

Rückgabewert

_dos_getftime und *_dos_setftime* liefern bei fehlerfreier Ausführung 0 zurück.

Im Fehlerfall wird der Fehlercode des Betriebssystems zurückgeliefert, und die globale Variable *errno* erhält einen der folgenden Werte:

EACCES	Zugriff verweigert
EBADF	Ungültiges Handle

Siehe auch

fstat, *stat*

_dos_gettime, _dos_settime — dos.h

Funktion

Liest und setzt die Systemzeit.

Syntax

```
void _dos_gettime(struct dostime_t *timep);
unsigned _dos_settime(struct dostime_t *timep);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung

_dos_gettime speichert die aktuelle Systemzeit in der über *timep* angegebenen Struktur des Typs *dostime_t*.

_dos_settime setzt die Systemzeit auf die Werte in der Struktur des Typs *dostime_t*, auf die *timep* zeigt.

Der Strukturtyp *dostime_t* ist folgendermaßen definiert:

```
struct dostime_t {
   unsigned char hour;      /* Stunden (0 bis 23) */
   unsigned char minute;    /* Minuten (0 bis 59) */
   unsigned char second;    /* Sekunden (0 bis 59) */
   unsigned char hsecond;   /* Hundertstel Sekunden */
                            /* (0 bis 99 */
};
```

Rückgabewert *_dos_gettime* hat keinen Rückgabewert.

Bei fehlerfreier Ausführung liefert *_dos_settime* als Ergebnis 0. Andernfalls wird der Fehlercode des Betriebssystems zurückgeliefert, und die globale Variable *errno* erhält den Wert:

EINVAL Ungültige Uhrzeit

Siehe auch *_dos_getdate*, *_dos_setdate*, *_dos_settime*, *stime*, *time*

_dos_getvect — dos.h

Funktion Liest einen Interrupt-Vektor.

Syntax

```
void interrupt(*_dos_getvect(unsigned interruptno))();
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung Die Prozessoren der 8086-Familie reservieren eine Reihe von Interrupt-Vektoren, die von 0 bis 255 durchnumeriert sind. Der vier Byte lange Wert jedes Vektors ist die Adresse einer Interrupt-Funktion.

_dos_getvect liefert den Wert des Vektors *interruptno* als **far**-Zeiger auf eine Funktion des Typs **interrupt** zurück. Der Wert von *interruptno* muß zwischen 0 und 255 liegen.

Rückgabewert *_dos_getvect* liefert den aktuellen vier Byte langen Wert zurück, der im Interrupt-Vektor *interruptno* gespeichert ist.

Siehe auch *_disable*, *_enable*, *_dos_setvect*

_dos_open — fcntl.h, share.h, dos.h

Funktion Öffnet eine Datei für Lese- oder Schreibaktionen.

Syntax

```
unsigned_dos_open(const char *filename, unsigned oflags int *handlep);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung

_dos_open öffnet die durch *filename* angegebene Datei entsprechend des Parameters *oflags* für Lese- und/oder Schreibzugriffe. Die Datei wird stets im Binärmodus geöffnet. *_dos_open* speichert das Datei-Handle an der von *handlep* angegebenen Adresse.

oflags verwendet die Flags der folgenden beiden Listen. Aus der ersten Liste darf (und muß) genau ein Flag verwendet werden. Die übrigen Flags können in jeder logischen Kombination benutzt werden.

Liste 1: Lese-/Schreib-Flags

O_RDONLY	Nur Lesen
O_WRONLY	Nur Schreiben
O_RDWR	Lesen und Schreiben

Die folgenden zusätzlichen Werte können mit einer ODER-Operation mit in *oflags* aufgenommen werden:

Diese symbolischen Konstanten sind in fcntl.h und in share.h definiert.

Liste 2: Weitere Zugriffs-Flags

O_NOINHERIT	Datei wird nicht an Kindprozesse vererbt.
SH_COMPAT	Ermöglicht weitere Zugriffe auf diese Datei mit SH_COMPAT. Wurde die Datei bereits in einem gemeinsamen Zugriffsmodus geöffnet, ergibt der Aufruf einen Fehler.
SH_DENYRW	Ermöglicht den Zugriff auf die Datei nur über das aktuelle Handle.
SH_DENYWR	Über andere Handles kann diese Datei nur gelesen, aber nicht beschrieben werden.
SH_DENYRD	Über andere Handles kann diese Datei nur beschrieben werden.
SH_DENYNO	Sämtliche Zugriffe auf diese Datei sind erlaubt, jedoch nicht solche mit SH_COMPAT.

In einem Aufruf von *_dos_open* kann jeweils nur eine der SH_DENYxxx-Konstanten angegeben werden. Diese Zugriffsattribute sind zusätzlich zu bestehenden Dateisperrungen gültig.

Die maximal mögliche Anzahl offener Dateien wird durch HANDLE_MAX definiert.

Rückgabewert

_dos_open liefert bei fehlerfreier Ausführung 0 zurück und speichert das Datei-Handle an der durch *handlep* angegebenen Adresse. Der Dateizeiger – er markiert die aktuelle Position in der Datei – wird auf den Dateianfang gesetzt.

Im Fehlerfall liefert *_dos_open* den DOS-Fehlercode zurück. Die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht gestattet
EINVACC	Ungültiger Zugriffscode
ENOENT	Pfad oder Datei nicht gefunden
EMFILE	Zu viele Dateien offen

Siehe auch *open*, *_rtl_read*, *sopen*

D

_dos_read io.h, dos.h

Funktion Liest Daten aus einer Datei.

Syntax

```
unsigned _dos_read(int handle, void far *buf, unsigned *nread);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_read* liest *len* Bytes aus der durch *handle* bezeichneten Datei in den durch den **far**-Zeiger *buf* angegebenen Speicherbereich. Die Anzahl der gelesenen Bytes wird an der Adresse gespeichert, auf die *nread* zeigt. Im Fehlerfall oder beim Erreichen des Dateiendes kann diese Zahl kleiner als *len* sein.

Die Funktion *_dos_read* entfernt keine Wagenrücklaufzeichen (CR), da sie ausschließlich mit binären Dateien arbeitet.

Der Parameter *handle* ist das über einen Aufruf von *_dos_creat*, _dos_creatnew, oder *_dos_open* erhaltene Datei-Handle.

Bei Diskettendateien beginnt der Lesevorgang ab der aktuellen Position des Dateizeigers. Nach Beendigung des Lesevorgangs wird der Dateizeiger um die Anzahl der gelesenen Bytes erhöht. Bei Gerätedateien ist eine Position nicht definiert – hier liest *_dos_read* direkt vom entsprechenden Gerät.

_dos_read kann maximal UINT_MAX-1 Bytes lesen, da UINT_MAX selbst dem Wert -1 und damit dem Fehlerindikator entspricht. Die Konstante UINT_MAX ist in limits.h definiert.

Rückgabewert Bei fehlerfreier Ausführung liefert *_dos_read* den Wert 0 zurück. Andernfalls liefert die Funktion den DOS-Fehlercode zurück und setzt die globale Variable *errno* auf einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
EBADF	Ungültiges Handle

Siehe auch *_rtl_open*, *read*, *_rtl_write*

_dos_setdate

Siehe *_dos_getdate*.

_dos_setdrive

Siehe *_dos_getdrive*.

_dos_setfileattr

Siehe *_dos_getfileattr*.

_dos_setftime

Siehe *_dos_getftime*.

_dos_settime

Siehe *_dos_gettime*.

_dos_setvect — dos.h

Funktion Setzt den Eintrag eines Interrupt-Vektors.

Syntax

```
void _dos_setvect(unsigned interruptno, void interrupt (*isr) ());
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung Die Prozessoren der 8086-Familie reservieren eine Reihe von Interrupt-Vektoren, die von 0 bis 255 durchnumeriert sind. Der vier Byte lange Wert jedes Vektors ist die Adresse einer Interrupt-Funktion.

_dos_setvect setzt den Interrupt-Vektor Nummer *interruptno* so, daß er auf die C-Funktion zeigt, die durch den **far**-Zeiger *isr* (interrupt service routine) angegeben wird. Diese C-Funktion muß als **interrupt**-Routine deklariert worden sein.

Solange die in dos.h deklarierten Prototypen verwendet werden, ist *_dos_setvect* ohne weitere Modifikationen in jedem Speichermodell anwendbar.

Rückgabewert Keiner.

Siehe auch *_dos_getvect*

dostounix — dos.h

Funktion Konvertiert Datum und Uhrzeit in das UNIX-Format.

Syntax

```
long dostounix(struct date *d, struct time *t);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *dostounix* erwartet einen Zeiger auf eine Struktur des Typs *date* (die beispielsweise von *getdate* zurückgegeben wird) sowie einen Zeiger auf eine Struktur des Typs *time* (wie sie *gettime* liefert) und konvertiert beide Strukturen in das UNIX-Datumsformat. Die Strukturen *date* und *time* müssen Datum und Uhrzeit im DOS-Format enthalten.

Die Datums- und die Zeitangabe dürfen nicht älter sein als der 1. Januar 1980, 00:00:00 Uhr.

Rückgabewert *dostounix* liefert Datum und Uhrzeit im UNIX-Format zurück: die Zahl der Sekunden seit dem 1. Januar 1970, 00:00:00 Uhr (GMT – Greenwich Mean Time).

Siehe auch *getdate, gettime, unixtodos*

_dos_write — dos.h

Funktion Schreibt Daten in eine Datei.

Syntax

```
unsigned _dos_write(int handle, const void __far *buf, unsigned len,
                    unsigned *nwritten);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *_dos_write* liest mit Hilfe der DOS-Funktion 0x40 *len* Bytes aus dem Puffer, auf den der **__far**-Zeiger *buf* zeigt, und schreibt sie in die durch *handle* bezeichnete Datei. Da die Funktion *_dos_write* nur mit binären Dateien arbeitet, übersetzt sie Zeilenvorschubzeichen (LF) nicht in CR/LF (Wagenrücklauf/Zeilenvorschub).

_dos_write speichert die Anzahl der geschriebenen Bytes an der durch *nwritten* bezeichneten Adresse. Werden weniger als *len* Bytes geschrieben, sollte dies als Fehler interpretiert werden und deutet normalerweise auf eine volle Diskette/Festplatte hin. Bei Dateien auf Diskette/Festplatte schreibt *_dos_write* ab der aktuellen Position des Dateizeigers, bei Gerätedateien werden die Bytes direkt an das jeweilige Gerät gesendet.

Rückgabewert *_dos_write* liefert bei fehlerfreier Ausführung das Ergebnis 0. Andernfalls wird der Fehlercode des Betriebssystems zurückgeliefert, und die globale Variable *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht erlaubt
EBADF	Ungültiges Handle

Siehe auch *_dos_open*, *_dos_creat*, *_dos_read*

dup — io.h

Funktion Verdoppelt ein Handle für eine Datei.

Syntax

```
int dup(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *dup* erzeugt aus einem Datei-Handle ein zweites Handle, das mit dem Original folgende Gemeinsamkeiten hat:

- die offene Datei bzw. das Gerät;
- den Dateizeiger (ändert man den Dateizeiger des einen Handle, so ändert man auch den des anderen);
- die Zugriffsberechtigungen (Lesen, Schreiben, Lesen und Schreiben).

handle ist ein Datei-Handle, das man durch einen Aufruf von *_rtl_creat*, *creat*, *_rtl_open*, *open*, *dup* oder *dup2* erhält.

Rückgabewert *dup* liefert bei fehlerfreier Ausführung das neu erzeugte Datei-Handle als positiven Integerwert zurück. Im Fehlerfall liefert die Routine den Wert -1, und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EMFILE	Zu viele offene Dateien
EBADF	Ungültiges Handle

Siehe auch *close*, *creat*, *creatnew*, *creattemp*, *dup2*, *fopen*, *open*, *_rtl_close*, *_rtl_creat*, *_rtl_open*

dup2 — io.h

Funktion
Bildet das Datei-Handle (*oldhandle*) auf dem existierendem Datei-Handle (*newhandle*) ab.

Syntax
```
int dup2(int oldhandle, int newhandle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung
dup2 erzeugt aus einem Handle ein zweites, das mit dem Original folgende Gemeinsamkeiten hat:

- die offene Datei bzw. das Gerät;
- den Dateizeiger (ändert man den Dateizeiger des einen Handle, so ändert man auch den des anderen);
- die Zugriffsberechtigungen (Lesen, Schreiben, Lesen und Schreiben).

dup2 erzeugt ein neues Handle mit dem Wert von *newhandle*. Ist die mit *newhandle* verbundene Datei beim Aufruf von *dup2* geöffnet, wird sie geschlossen. *newhandle* und *oldhandle* sind Datei-Handles, die durch Aufrufe von *creat*, *open*, *dup* oder *dup2* erzeugt wurden.

Rückgabewert
dup2 liefert bei fehlerfreier Ausführung den Wert 0 zurück.

Im Falle eines Fehlers liefert die Routine den Wert -1, und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EBADF	Ungültiges Handle
EMFILE	Zu viele offene Dateien

Siehe auch
close, *creat*, *creatnew*, *creattemp*, *dup*, *fopen*, *open*, *_rtl_close*, *_rtl_creat*, *_rtl_open*

ecvt — stdlib.h

Funktion
Konvertiert eine Gleitkommazahl in einen String.

Syntax
```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung
ecvt konvertiert *value* in einen nullterminierten String mit *ndig* Ziffern, der mit der höchstwertigen Ziffer (links) beginnt. *ecvt* liefert einen Zeiger auf diesen String zurück. Über *dec* wird die Position des Dezimalpunkts relativ zum Beginn des Strings zurückgeliefert (negative Werte bedeuten hier: »Dezimalpunkt befindet sich links vom Stringanfang«). Der String selbst enthält keinen Dezimalpunkt. Wenn *value* negativ ist, dann setzt

	ecvt das über *sign* adressierte Wort auf einen Wert ungleich null (für positive Werte von *value* ist **sign* == 0). Die letzte Ziffer des zurückgelieferten Strings ist gerundet.
Rückgabewert	*ecvt* liefert einen Zeiger auf diesen statischen String zurück, dessen Inhalt durch jeden Aufruf von *fcvt* und *ecvt* überschrieben wird.
Siehe auch	*fcvt*, *gcvt*, *sprintf*

__emit__ dos.h

Funktion Fügt Literale direkt in den Objektcode ein.

Syntax

```
void __emit__(argument, ... );
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

__emit__ ist eine Inline-Funktion, die das direkte Einfügen von Literalen während der Compilierung des Objektcodes ermöglicht. Die Funktion dient zur Erzeugung von Assembler-Anweisungen ohne Verwendung eines Assemblers.

Im allgemeinen wird man als Argumente für *__emit__* einzelne Bytes (d.h. Prozessorbefehle) verwenden. Es lassen sich aber auch wesentlich komplexere Konstrukte bilden – inklusive Zugriffen auf C-Variablen.

➥ Sie sollten diese Funktion nur verwenden, wenn Sie mit der Maschinensprache der 80x86-Prozessorfamilie vertraut sind. Sie können mit Hilfe dieser Funktion beliebige Bytes in den Befehlscode einer Funktion einfügen. Ein einziges fehlerhaftes Byte kann zu unerwartetem Verhalten Ihres Programms und auch ohne weiteres zu einem Systemabsturz führen. Der Compiler unternimmt keinen Versuch, die eingefügten Byte-Folgen auf formale Korrektheit oder in irgendeiner anderen Weise zu prüfen. Der Compiler bemerkt nicht, ob Sie Maschinenbefehle einfügen, die den Inhalt von Prozessorregistern oder des Speichers ändern. Anders als bei der Verwendung von Inline-Assembler, wo beispielsweise die Benutzung der Register SI und DI erkannt wird, ergreift der Compiler keine Maßnahmen zur Sicherung von Registerinhalten. Der korrekte Einsatz dieser Funktion liegt allein in der Verantwortung des Programmierers.

__emit__ erwartet zumindest ein Argument, es können aber beliebig viele übergeben werden. Die Argumente für diese Funktion werden nicht wie Argumente bei normalen Funktionsaufrufen behandelt, da keinerlei Konvertierungen durchgeführt werden.

Die Parameter dieser Funktion unterliegen denselben Beschränkungen wie Initialisierungswerte statischer Objekte: es dürfen Integer- und Gleitkommakonstanten und die Adressen statischer Objekte verwendet werden. Die Werte dieser Ausdrücke werden exakt an der Stelle des Aufrufs in den Objektcode genauso geschrieben, als dienten sie der Initialisierung von Daten. Es darf auch die Adresse eines Parameters oder einer automatischen Variablen plus oder minus einem konstanten Offset verwendet werden. Für diese Argumente wird der Offset der Variablen vom BP-Register gespeichert.

Die Anzahl der im Objektcode gespeicherten Bytes hängt, abgesehen von folgenden Einschränkungen, vom Typ des jeweiligen Arguments ab:

E

- eine vorzeichenbehaftete Integer-Konstante (zum Beispiel: 0x90) wird als ein Zeichen behandelt, wenn sie im Bereich zwischen 0 und 255 liegt;
- für die Adresse einer automatischen Variablen oder eines Parameters wird ein Byte geschrieben, wenn der Offset der Variablen vom BP-Register zwischen -128 und 127 liegt, andernfalls wird ein Wort (2 Bytes) geschrieben.

Ein einfaches Byte schreiben Sie so:

```
__emit__ (0x90);
```

Falls Sie ein Wort einfügen wollen, dessen Wert kleiner als 255 ist, wandeln Sie den Typ einfach mit einer der beiden folgenden Methoden in **unsigned** um:

```
__emit__ (0xB8, (unsigned)17);
```

```
__emit__ (0xB8, 17u);
```

Zwei bzw. vier Byte lange Adressen können durch die Typumwandlung einer Adresse in **void near*** bzw. **void far*** erzwungen werden.

Rückgabewert Keiner.

enable, _enable

Siehe *disable.*

_endthread process.h

Funktion Beendet die Ausführung eines Threads.

Syntax

```
void _endthread(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung

Die Funktion *_endthread* beendet den aktuell ausgeführten Thread, der zuvor mit einem Aufruf von *_beginthread* oder *_beginthreadNT* gestartet wurde.

Diese Funktion ist nur in den Multi-Thread Bibliotheken verfügbar.

Rückgabewert

Keiner.

_beginthread, *_beginthreadNT*

eof io.h

Funktion

Prüft, ob das Ende einer Datei erreicht ist.

Syntax

```
int eof(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

eof prüft, ob der Dateizeiger der durch *handle* bezeichneten Datei am Dateiende steht.

Rückgabewert

Wenn das Ende der Datei erreicht ist, liefert *eof* den Wert 1, ansonsten den Wert 0. Der Rückgabewert -1 zeigt einen Fehler an; die globale Variable *errno* erhält in diesem Fall den Wert

EBADF Ungültiges Handle

Siehe auch

clearerr, *feof*, *ferror*, *perror*

exec... process.h

Funktion

Funktionen zum Laden und Starten eines anderen Programms.

Syntax

```
int execl(char *path, char *arg0, ..., *argn, NULL);
int execle(char *path, char *arg0, ..., *argn, NULL, char **env);

int execlp(char *path, char *arg0, ..., *argn, NULL);
int execlpe(char *path, char *arg0, ..., *argn, NULL, char **env);
```

```
int execv(char *path, char *argv[]);
int execve(char *path, char *argv[], char **env);

int execvp(char *path, char *argv[]);
int execvpe(char *path, char *argv[], char **env);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

Die Funktionen der *exec...*-Familie laden und starten andere Programme, die als *Kindprozesse* bezeichnet werden. Wenn ein *exec...*-Aufruf fehlerfrei ausgeführt werden konnte, überlagert der Kindprozeß (child process) seinen Elternprozeß (parent process). Im Hauptspeicher muß genügend Platz sein, um den Kindprozeß laden und ausführen zu können. Der Parameter *path* gibt den Dateinamen des Kindprozesses an – verwendet wird der Standard-Suchalgorithmus:

- Wenn weder eine Namenserweiterung noch ein Punkt angegeben ist, wird zuerst nach Dateien ohne Namenserweiterung gesucht. Wird die Datei nicht gefunden, so wird der Dateiname um .EXE erweitert und die Suche erneut gestartet. Wird die Datei wieder nicht gefunden, so wird .COM angehängt und abermals gesucht. Schließlich .BAT angehängt und ein letztes Mal gesucht. Zur Ausführung des Programmdatei wird der Kommandozeileninterpreter COMSPEC benutzt.
- Wenn eine Namenserweiterung oder ein Punkt angegeben ist, wird nur nach dem angegebenen Dateinamen gesucht.

Die einzelnen Funktionen der *exec...*-Familie unterscheiden sich durch die Namensendungen *l*, *v*, *p* und *e*, wobei jeder dieser Buchstaben auf andere Fähigkeiten hinweist.

Es bedeuten:

- *l* Die Zeiger (*arg0*, ..., *argn*) werden in Form einer Liste einzelner Argumente übergeben. Die Listenform wird normalerweise verwendet, wenn die Zahl der zu übergebenden Argumente bekannt und konstant ist.
- *v* Die Argumentzeiger werden als Zeiger-Array übergeben, das heißt als *argv[0]*, *argv[1] ...argv[n]*. Diese Form wird normalerweise bei einer variablen Zahl von Argumenten verwendet.
- *p* Die *exec*-Funktion sucht nicht nur innerhalb des aktuellen Verzeichnisses nach der Datei, sondern auch in sämtlichen in der Umgebungsvariablen PATH festgelegten Verzeichnissen, falls der Parameter *path* kein explizites Verzeichnis enthält.

- *e* Erlaubt die Übergabe des Zeiger-Array *env* (d.h. eines eigenen Environment für den Kindprozeß). Über *exec...*-Varianten ohne *e* aufgerufene Prozesse verwenden dieselben Environment-Parameter wie das aufrufende Programm.

Jede Variante der *exec...*-Familie enthält entweder ein *l* oder ein *v* im Namen, die weiteren Kombinationen mit *p* und *e* sind optional.

Zwei Beispiele dazu:

- *execl* verwendet eine Liste von Argumenten, sucht nur das aktuelle Verzeichnis oder das Stammverzeichnis nach dem Kindprozeß ab und übergibt die Environment-Parameter, die auch für das momentan laufende Programm gültig sind;
- *execvpe* übernimmt ein Zeiger-Array, führt nötigenfalls eine Suche nach dem Kindprozeß über die durch PATH festgelegten Pfade aus und übergibt Environment-Parameter, die vom Programmierer für diesen Prozeß definiert worden sind.

Sämtliche *exec...*-Funktionen müssen zumindest ein Argument an den Kindprozeß übergeben, nämlich *arg0* bzw. *argv[0]*. Dieses Argument enthält per Konvention eine Kopie von *path*. Die Übergabe anderer Daten als erstem Parameter widerspricht nur den Gepflogenheiten, ist aber kein Fehler.

Somit ist das Argument *path* für den Kindprozeß verfügbar.

Bei den *l*-Funktionen zeigt *arg0* normalerweise auf *path*. Die Parameter *arg1,...,argn* bilden die aus weiteren Strings bestehende neue Argumentliste, die mit einer NULL nach *argn* beendet werden muß.

Die *e*-Funktionen erwarten das Argument *env* als Zeiger-Array. Jedes Element dieses Array zeigt auf einem nullterminierten String der Form

```
U_Var = Wert
```

wobei *U_Var* der Name einer Umgebungsvariablen ist und *Wert* der String, auf den *U_Var* gesetzt wird.

Das letzte Element des Array muß einen Zeiger mit dem Wert NULL enthalten. Dadurch wird das Listenende gekennzeichnet. Wenn *env* den Wert NULL hat, verwendet der Kindprozeß das Environment seines Elternprozesses.

Die Gesamtlänge sämtlicher als *arg* übergebenen Argumente darf bei 16 Bit Anwendungen 128 Byte, bei Win32 Anwendungen 260 Byte nicht überschreiten (inklusive der automatisch eingesetzten Leerzeichen zur Trennung einzelner Parameter). Nullzeichen und Environment-Parameter werden bei dieser Zählung nicht berücksichtigt.

exec... schließt keine Dateien – sämtliche durch den Elternprozeß geöffneten Dateien stehen auch dem Kindprozeß zur Verfügung.

Rückgabewert

Bei fehlerfreier Ausführung kehrt eine *exec...*-Funktion überhaupt nicht zum Programm zurück. Konnte die *exec...*-Funktion dagegen nicht ausgeführt werden, dann wird -1 zurückgeliefert, und die globale Variable *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht erlaubt
EMFILE	Zu viele offene Dateien
ENOENT	Pfad/Dateiname nicht gefunden
ENOEXEC	*exec*-Formatfehler
ENOMEM	Nicht genug Platz im Hauptspeicher

Siehe auch

abort, *atexit*, *_exit*, *exit*, *_fpreset*, *searchpath*, *spawn...*, *system*

E

_exit stdlib.h

Funktion

Beendet das laufende Programm.

Syntax

```
void _exit(int status);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

_exit beendet das Programm, ohne Dateien zu schließen, Ausgabepuffer zu leeren oder Exit-Funktionen aufzurufen.

_exit beendet das Programm mit dem als *status* angegebenen Exit-Code, der vom aufrufenden Prozeß geprüft werden kann. Der Wert 0 steht normalerweise für eine fehlerfreie Ausführung, Werte ungleich null signalisieren Fehler.

Rückgabewert

Keiner.

Siehe auch

abort, *atexit*, *exec...*, *exit*, *spawn...*

exit stdlib.h

Funktion

Beendet das laufende Programm.

Syntax

```
void exit(int status);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

exit beendet das aufrufende Programm. Vor dem Rücksprung werden sämtliche Dateipuffer geschrieben, offene Dateien geschlossen und (mit *atexit*) festgelegte Exit-Funktionen aufgerufen.

Die Funktion beendet das Programm mit dem als *status* angegebenen Exit-Code, der vom aufrufenden Prozeß geprüft werden kann. Der Wert 0 steht normalerweise für eine fehlerfreie Ausführung, Werte ungleich null signalisieren Fehler. Folgende Konstanten stehen für *status* zur Verfügung:

EXIT_FAILURE

Fehlerhaftes Programmende – Signal an das Betriebssystem, daß das Programm mit einem Fehler endete.

EXIT_SUCCESS

Normales Programmende.

Rückgabewert Keiner.

Siehe auch *abort*, *atexit*, *exec...*, *_exit*, *signal*, *spawn...*

exp, expl — math.h

Funktion Berechnet die Exponentialfunktion e^x.

Syntax

```
double exp(double x);
long double expl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
exp	■	■	■	■	■	■	■
expl	■		■	■			■

Beschreibung *exp* berechnet die Exponentialfunktion e^x.

expl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert *exp* liefert e^x zurück. Falls die Argumente zu nicht berechenbaren Resultaten führen, werden die folgenden Fehlerbedingungen erzeugt: Bei einem Überlauf liefert *exp* HUGE_VAL und *expl* _LHUGE_VAL. Die globale Variable *errno* wird in diesen Fällen auf folgenden Wert gesetzt:

ERANGE Ergebnis außerhalb des Rechenbereichs

Bei einem Unterlauf liefern *exp* und *expl* 0,0 zurück, die globale Variable *errno* wird nicht verändert.

Die Fehlerbehandlung für *exp* und *expl* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

Siehe auch *frexp*, *ldexp*, *log*, *log10*, *_matherr*, *pow*, *pow10*, *sqrt*

_expand — malloc.h

Funktion Vergrößert oder verkleinert einen Heap-Speicherblock.

Syntax

```
void *_expand(void *block, size_t size);
```

OS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung Diese Funktion versucht die Größe des bereits reservierten Speicherblocks *block* zu ändern, ohne ihn dabei zu verschieben. Die im Speicherblock enthaltenen Daten werden bis hin zum Minimum der alten und neuen Bereichsgröße nicht geändert. Der Speicherblock muß durch einen vorhergehenden Aufruf von *malloc, calloc,* oder *realloc* reserviert und darf noch *nicht* wieder freigegeben worden sein.

Rückgabewert Konnte die Größe des Speicherblocks ohne Verschiebung geändert werden, so liefert *_expand* die (unveränderte) Adresse des Speicherblocks zurück. Andernfalls wird ein NULL-Zeiger zurückgeliefert und die Größe des Speicherbereichs bleibt unverändert.

Siehe auch *malloc, calloc, realloc*

F

fabs, fabsl — math.h

Funktion Berechnet den absoluten Wert einer Gleitkommazahl.

Syntax

```
double fabs(double x);
long double fabsl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
fabs	■	■	■	■	■	■	■
fabsl	■		■	■			■

Beschreibung *fabs* berechnet den absoluten Betrag des **double**-Arguments *x*. *fabsl* ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double**.

Rückgabewert *fabs* und *fabsl* liefern den absoluten Betrag von *x* zurück.

Siehe auch *abs, cabs, labs*

farcalloc — alloc.h

Funktion Reserviert Speicherplatz auf dem FAR-Heap.

Syntax

```
void far *farcalloc(unsigned long nunits, unsigned long unitsz);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

farcalloc reserviert Platz auf dem FAR-Heap für ein Array mit *nunits* Elementen, deren Größe jeweils *unitsz* Bytes beträgt.

Bei der Arbeit mit dem FAR-Heap sind einige sonst vorhandene Grenzen aufgehoben:

- der gesamte verfügbare RAM-Speicherplatz kann reserviert werden;
- die Größe einzelner Blöcke kann 64 KByte überschreiten;
- für Zugriffe auf die Blöcke müssen grundsätzlich **far**-Zeiger (oder **huge**-Zeiger, wenn die Blöcke größer als 64 KByte sind) benutzt werden.

Bei Verwendung der Speichermodelle Compact oder Large verhält sich *farcalloc* ähnlich wie *calloc*, die Funktionen sind aber nicht identisch: die Parameter von *calloc* sind vom Typ **unsigned**, diejenigen von *farcalloc* dagegen vom Typ **unsigned long**.

Rückgabewert

farcalloc liefert einen **far**-Zeiger auf die Startadresse des neu reservierten Speicherblocks zurück. Wenn nicht ausreichend Platz zur Verfügung steht, wird der Zeigerwert NULL zurückgeliefert.

Siehe auch

calloc, *farfree*, *farmalloc*, *malloc*

farfree — alloc.h

Funktion

Gibt einen Speicherblock auf dem FAR-Heap frei.

Syntax

```
void farfree(void far *block);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

farfree gibt einen zuvor reservierten Speicherblock auf dem FAR-Heap wieder frei.

In den Modellen *Small* und *Medium* sind der Heap und der FAR-Heap zwei völlig voneinander getrennte Speicherbereiche – durch *farmalloc* reservierte Blöcke können nicht mit *free* freigegeben werden, durch *malloc* reservierte Blöcke nicht mit *farfree*.

Rückgabewert

Keiner.

Siehe auch

farcalloc, *farmalloc*

farmalloc alloc.h

Funktion Reserviert Speicher auf dem FAR-Heap.

Syntax

```
void far *farmalloc(unsigned long nbytes);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *farmalloc* reserviert einen Speicherblock von *nbytes* Größe auf dem FAR-Heap.

Bei der Arbeit mit dem FAR-Heap sind einige sonst vorhandene Grenzen aufgehoben:

- der gesamte verfügbare RAM-Speicherplatz kann reserviert werden;
- die Größe einzelner Blöcke kann 64 KByte überschreiten;
- für Zugriffe auf die Blöcke müssen grundsätzlich **far**-Zeiger benutzt werden.

Bei Verwendung der Speichermodelle Compact oder Large verhält sich *farmalloc* ähnlich wie die Funktion *malloc* – identisch sind sie aber nicht: Die Parameter von *malloc* sind vom Typ **unsigned**, die von *farmalloc* dagegen vom Typ **unsigned long**.

Rückgabewert *farmalloc* liefert einen **far**-Zeiger auf die Startadresse des neu reservierten Speicherblocks zurück. Wenn nicht genügend Platz zur Verfügung steht, wird der Zeigerwert NULL zurückgeliefert.

Siehe auch *farcalloc*, *farfree*, *farrealloc*, *malloc*

farrealloc alloc.h

Funktion Vergrößert oder verkleinert einen auf dem FAR-Heap reservierten Speicherbereich.

Syntax

```
void far *farrealloc(void far *oldblock, unsigned long nbytes);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *farrealloc* verändert die Größe eines bereits reservierten Blocks auf *nbytes*, wobei der Blockinhalt nötigenfalls in einen anderen Speicherbereich kopiert wird.

Bei der Arbeit mit dem FAR-Heap sind einige sonst vorhandene Grenzen aufgehoben:

F

- der gesamte verfügbare RAM-Speicherplatz kann reserviert werden;
- die Größe einzelner Blöcke kann 64 KByte überschreiten;
- für Zugriffe auf die Blöcke müssen grundsätzlich **far**-Zeiger benutzt werden.

Rückgabewert *farrealloc* liefert einen **far**-Zeiger auf die Startadresse des veränderten Speicherblocks zurück, die sich von der alten Startadresse unterscheiden kann. Falls nicht ausreichend Platz zur Verfügung steht, wird der Wert NULL geliefert.

Siehe auch *farmalloc*, *realloc*

fclose — stdio.h

Funktion Schließt einen Stream.

Syntax

```
int fclose(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *fclose* schließt den durch *stream* angegebenen Stream, wobei alle zum Stream gehörenden Puffer zuvor auf Diskette bzw. Platte geschrieben werden. Durch das System reservierte Puffer werden durch *fclose* freigegeben. Puffer, die mit *setbuf* oder *setvbuf* zugeordnet wurden, werden nicht automatisch freigegeben (es sei denn, beim Aufruf von *setvbuf* wurde als Pufferzeiger der Wert NULL übergeben).

Rückgabewert *fclose* liefert bei fehlerfreier Ausführung den Wert 0, im Fehlerfall den Wert EOF zurück.

Siehe auch *close*, *fcloseall*, *fdopen*, *fflush*, *flushall*, *fopen*, *freopen*

fcloseall — stdio.h

Funktion Schließt offene Streams.

Syntax

```
int fcloseall(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *fcloseall* schließt sämtliche offenen Streams außer *stdin*, *stdout*, *stderr*, *stdprn* und *stdaux*. Die Streams *stdprn* und *stdaux* sind unter OS/2 und Win32 nicht verfügbar.

Rückgabewert *fcloseall* liefert bei fehlerfreier Ausführung die Anzahl der geschlossenen Streams zurück, im Fehlerfall den Wert EOF.

Siehe auch *fclose, fdopen, flushall, fopen, freopen*

fcvt — stdlib.h

Funktion Konvertiert eine Gleitkommazahl in einen String.

Syntax

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *fcvt* konvertiert *value* in einen nullterminierten String mit *ndig* Ziffern rechts des Dezimalpunkts, der mit der höchstwertigen Ziffer (links) beginnt. *fcvt* liefert einen Zeiger auf diesen String zurück. Über *dec* wird die Position des Dezimalpunkts relativ zum Beginn des Strings zurückgeliefert (negative Werte bedeuten hier: »Dezimalpunkt befindet sich links vom Stringanfang«). Der String selbst enthält keinen Dezimalpunkt. Wenn *value* negativ ist, dann setzt *fcvt* das über *sign* adressierte Wort auf einen Wert ungleich null (für positive Werte von *value* ist **sign* == 0).

Die Ziffern werden auf *ndig* Stellen hinter dem Dezimalpunkt gerundet.

Rückgabewert *fcvt* liefert einen Zeiger auf einen statischen String zurück, dessen Inhalt durch jeden Aufruf dieser Funktion (oder von *ecvt*) neu gesetzt wird.

Siehe auch *ecvt, gcvt, sprintf*

fdopen — stdio.h

Funktion Ordnet einem Datei-Handle einen Stream zu.

Syntax

```
FILE *fdopen(int handle, char *type);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *fdopen* ordnet dem offenen Datei-Handle *handle*, das über einen Aufruf *creat, dup, dup2* oder *open* erhalten wurde, einen Stream zu. Der Typ des Stream muß zum Modus des Datei-Handle passen. Der Parameter *type* kann einen der folgenden Werte annehmen:

Wert	Beschreibung
r	Öffnung ausschließlich für Leseoperationen.
w	Erzeugung einer Datei für Schreiboperationen.
a	Öffnung für Schreiboperationen. Falls die Datei bereits existiert, werden zusätzliche Informationen an das aktuelle Dateiende angefügt, ansonsten wird sie für Schreiboperationen erzeugt.
r+	Öffnung einer Datei für Lese- und Schreiboperationen. Die Datei muß bereits existieren.
w+	Erzeugung einer neuen Datei für Lese- und Schreiboperationen.
a+	Öffnung einer Datei zum Lesen und Anhängen neuer Daten durch Schreiboperationen. Falls die Datei bereits existiert, bleibt ihr alter Inhalt erhalten – falls nicht, wird sie automatisch erzeugt.

Bei der Öffnung kann zusätzlich explizit angegeben werden, ob die Datei im Text- (t) oder Binärmodus (b) bearbeitet werden soll. Die Angabe geschieht durch ein einfaches Anhängen des jeweiligen Buchstabens (also *wb*, *w+b*, *rt*, *a+t* etc.).
Wenn die Modusangabe fehlt, wird der jeweilige Modus durch die globale Variable *_fmode* bestimmt, deren Wert über die direkte Zuweisung einer der in fcntl.h definierten Konstanten O_TEXT oder O_BINARY festgelegt werden kann.
Mit *r+* und *w+* erzeugte Dateien können sowohl gelesen als auch beschrieben werden – zwischen einem Wechsel von »Schreiben« nach »Lesen« muß allerdings ein Aufruf von *fseek* oder *rewind* stattfinden. Zwischen einem Wechsel von »Lesen« nach »Schreiben« muß ebenfalls ein Aufruf von *fseek* oder *rewind* stattfinden, es sei denn, die Leseoperation hat das Dateiende erreicht.

Rückgabewert

fdopen liefert bei fehlerfreier Ausführung einen Zeiger auf den neu geöffneten Stream zurück, im Fehlerfall den Wert NULL.

Siehe auch

fclose, *fopen*, *freopen*, *open*

feof — stdio.h

Funktion

Prüft, ob das Ende eines Stream erreicht ist.

Syntax

```
int feof(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

Das Makro *feof* prüft das Dateiende-Flag des über *stream* angegebenen Stream. Dieses Flag wird beim Erreichen des Dateiendes automatisch gesetzt. Leseoperationen liefern dann solange dieses Flag, bis *rewind* aufge-

rufen oder die Datei geschlossen wird. Vor jeder Schreiboperation in diese Datei wird das Flag gelöscht.

Rückgabewert *feof* liefert einen Wert ungleich 0 zurück, wenn das Dateiende-Flag des angegebenen Stream gesetzt ist, ansonsten ist der Rückgabewert 0.

Siehe auch *clearerr*, *eof*, *ferror*, *perror*

ferror stdio.h

Funktion Prüft Fehlerbedingungen bei Streams.

Syntax

```
int ferror(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *ferror* ist ein Makro, das den über *stream* angegebenen Stream auf Schreib- und Lesefehler prüft. Das Fehler-Flag eines Stream bleibt nach einem Fehler solange gesetzt, bis entweder *clearerr* oder *rewind* aufgerufen oder der Stream geschlossen wird.

Rückgabewert *ferror* liefert einen Wert ungleich null zurück, wenn das Fehler-Flag des angegebenen *stream* gesetzt ist, sonst den Wert 0.

Siehe auch *clearerr*, *eof*, *feof*, *fopen*, *gets*, *perror*

fflush stdio.h

Funktion Schreibt den Ausgabepuffer eines Stream in die zugeordnete Datei.

Syntax

```
int fflush(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *fflush* erzwingt bei gepufferten Ausgabe-Streams ein Schreiben eventueller Pufferinhalte, sorgt also dafür, daß ausgegebene Daten in die zugehörige Datei geschrieben werden.

Der durch *stream* bezeichnete Stream bleibt offen. Auf ungepufferte Streams hat *fflush* keine Auswirkung.

Rückgabewert Bei fehlerfreier Ausführung liefert *fflush* den Wert 0, im Fehlerfall den Wert EOF zurück.

Siehe auch *fclose*, *flushall*, *setbuf*, *setvbuf*

fgetc stdio.h

Funktion Liest ein Zeichen aus einem Stream.

Syntax

```
int fgetc(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *fgetc* liest das jeweils nächste Zeichen aus dem Eingabe-Stream *stream*.

Rückgabewert *fgetc* konvertiert bei fehlerfreier Ausführung das gelesene Zeichen in einen **int**-Wert ohne Vorzeichenerweiterung und liefert diesen zurück. Bei Erreichen des Dateiendes oder im Fehlerfall wird der Wert EOF zurückgeliefert.

Siehe auch *fgetchar, fputc, getc, getch, getchar, getche, ungetc, ungetch*

fgetchar stdio.h

Funktion Liest ein Zeichen aus *stdin*.

Syntax

```
int fgetchar(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung Die Funktion *fgetchar* liest das jeweils nächste Zeichen aus *stdin* und liefert es zurück. Sie ist definiert als *fgetc(stdin)*.

➥ In Win32 oder Win32s GUI-Anwendungen muß *stdin* umgelenkt werden.

Rückgabewert *fgetchar* konvertiert bei fehlerfreier Ausführung das gelesene Zeichen in einen **int**-Wert ohne Vorzeichenerweiterung und liefert diesen zurück. Bei Erreichen des Dateiendes oder im Fehlerfall wird der Wert EOF zurückgeliefert.

Siehe auch *fgetc, fputchar, freopen, getchar*

fgetpos stdio.h

Funktion Ermittelt die aktuelle Position des Dateizeigers einer Datei.

Syntax

```
int fgetpos(FILE *stream, fpos_t *pos);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *fgetpos* speichert die Position des zu *stream* gehörigen Dateizeigers in **pos*. Der Wert wird speziell codiert gespeichert und ist nur als Parameter für einen folgenden Aufruf von *fsetpos* sinnvoll.

Rückgabewert *fgetpos* liefert bei fehlerfreier Ausführung 0 zurück. Im Fehlerfall liefert *fgetpos* einen Wert ungleich null und setzt die globale Variable *errno* auf:

EBADF	Ungültiges Handle
EINVAL	Ungültiges Argument

Siehe auch *fseek, fsetpos, ftell, tell*

fgets — stdio.h

Funktion Liest einen String aus einem Stream.

Syntax

```
char *fgets(char *s, int n, FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *fgets* liest einen String aus *stream* in den durch *s* angegebenen String und bricht ab, wenn entweder ein Zeilenvorschub (\n) gelesen wird oder n-1 Zeichen gelesen wurden. Der Zeilenvorschub wird am Ende von s gespeichert. Anschließend hängt *fgets* automatisch ein Nullzeichen (\0) an, um das Ende des Strings zu markieren.

Rückgabewert *fgets* liefert bei fehlerfreier Ausführung den durch s bezeichneten String zurück. Bei Erreichen des Dateiendes oder im Fehlerfall ist der Rückgabewert NULL.

Siehe auch *cgets, fputs, gets*

filelength — io.h

Funktion Ermittelt die Länge einer Datei in Bytes.

Syntax

```
long filelength(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *filelength* liefert die Länge der über *handle* angegebenen Datei in Bytes zurück.

Rückgabewert Bei fehlerfreier Ausführung liefert *filelength* die Dateigröße in Bytes als **long**-Wert. Im Fehlerfall liefert *filelength* -1 zurück und setzt die globale Variable *errno* auf

EBADF	Ungültiges Handle

Siehe auch *fopen*, *lseek*, *open*

fileno — stdio.h

Funktion Liefert das zu einem Stream gehörende Datei-Handle.

Syntax

```
int fileno(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *fileno* ist ein Makro und liefert das Handle des über den Parameter *stream* angegebenen Stream zurück. Wenn *stream* mehr als ein Handle zugeordnet ist, wird das beim ersten Öffnen des Stream vergebene Handle zurückgeliefert.

Rückgabewert *fileno* liefert das zu *stream* gehörende Datei-Handle als Integer zurück.

Siehe auch *fdopen*, *fopen*, *freopen*

findfirst — dir.h

Funktion Durchsucht ein Dateiverzeichnis.

Syntax

```
int findfirst(const char *pathname, struct ffblk *ffblk, int dos.h attrib);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *findfirst* startet die Suche in einem Verzeichnis nach Dateien, die über Attribute oder Jokerzeichen angegeben werden.

Der Parameter *pathname* ist ein String und bezeichnet den zu suchenden Eintrag. Er kann optional eine Laufwerksbezeichnung, einen Pfad und den zu suchenden Dateinamen enthalten. Nur im Dateinamen sind Jokerzeichen (* und ?) erlaubt. Falls eine passende Datei gefunden wird, werden in der über *ffblk* angegebenen Struktur die im Verzeichnis enthaltenen Dateiinformationen gespeichert.

Die Struktur *ffblk* ist folgendermaßen definiert:

32

```
struct ffblk {
   char ff_reserved[21];  /* von DOS verwendet */
   char ff_attrib;        /* gefundenes Attribut */
   int  ff_ftime;         /* Uhrzeit der letzten Änderung */
   int  ff_fdate;         /* Datum der letzten Änderung */
   long ff_fsize;         /* Dateigröße in Bytes */
   char ff_name[13];      /* Name der gefundenen Datei */
};
```

16

```
struct ffblk {
   long           ff_reserved;
   long           ff_fsize;     /* Dateigröße */
   unsigned long  ff_attrib;    /* gefundenes Attribut */
   unsigned short ff_ftime;     /* Uhrzeit der letzten Änderung */
   unsigned short ff_fdate;     /* Datum der letzten Änderung */
   char           ff_name[256]; /* Name der gefundenen Datei */
};
```


Über den Parameter *attrib* wird das Attribut des zu suchenden Eintrags angegeben. *attrib* kann eine der folgenden Konstanten sein, die in dos.h definiert sind:

FA_RDONLY	Nur-Lesen-Attribut
FA_HIDDEN	Verborgene Datei
FA_SYSTEM	Systemdatei
FA_LABEL	Datenträgername
FA_DIREC	Verzeichnis
FA_ARCH	Archiv-Bit

Eine Kombination obiger Konstanten kann durch eine ODER-Verknüpfung angegeben werden.

Ausführliche Informationen über diese Attribute finden Sie in den Referenzhandbüchern Ihres Betriebssystems.

Beachten Sie, daß *ff_ftime* und *ff_fdate* Bitfelder für die aktuelle Uhrzeit und das aktuelle Datum enthalten. Die Struktur dieser Bitfelder ist vom Betriebssystem vorgegeben. Es handelt sich in beiden Fällen um 16 Bit große Strukturen, die in drei Felder unterteilt sind:

ff_ftime:

Bits 0 bis 4	Resultierende Sekunden geteilt durch 2 (d.h. 10 bedeutet hier 20 Sekunden)
Bits 5 bis 10	Minuten
Bits 11 bis 15	Stunden

ff_fdate:

Bits 0 bis 4	Tag
Bits 5 bis 8	Monat
Bits 9 bis 15	Jahre seit 1980 (d.h. 9 bedeutet hier 1989)

Die in io.h definierte Struktur *ftime* verwendet Datums und Uhrzeit-Bitfelder ähnlicher Struktur wie in *ff_fdate* und *ff_ftime*.

Rückgabewert

findfirst liefert den Wert 0 zurück, wenn ein zu *pathname* passender Eintrag gefunden wurde. Wenn kein Eintrag gefunden wird oder der angegebene Dateiname aus irgendeinem Grund ungültig ist, wird -1 als Ergebnis geliefert und die globale Variable *errno* erhält den Wert:

ENOENT Datei/Pfad nicht gefunden

_doserrno wird in diesem Fall auf einen der folgenden Werte gesetzt:

ENMFILE Keine weiteren passenden Einträge
ENOENT Datei/Pfad nicht gefunden

Siehe auch

findnext, *getftime*, *setftime*

findnext — dir.h

Funktion

Setzt eine *findfirst*-Suche fort.

Syntax

```
int findnext(struct ffblk *ffblk);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

Mit *findnext* kann ein Verzeichnis nach weiteren Einträgen abgesucht werden, die zum Parameter *pathname* einer vorangegangenen *findfirst*-Suche passen. Als Argument muß die Struktur des Typs *ffblk* übergeben werden, die durch *findfirst* gefüllt wurde, da diese notwendige Informationen zur Fortsetzung der Suche enthält. *findnext* liefert bei jedem Aufruf einen weiteren Dateieintrag über *ffblk* zurück, bis das Verzeichnis keine weiteren Einträge mehr enthält, die dem Parameter *pathname* des vorherigen Aufrufs von *findfirst* entsprechen.

Rückgabewert

findnext liefert den Wert 0 zurück, wenn ein weiterer, zu *pathname* passender, Eintrag gefunden wurde. Wenn keine weiteren Einträge mehr vorhanden sind oder der angegebene Dateiname aus irgendeinem Grund ungültig ist, wird -1 geliefert und *errno* erhält den Wert:

ENOENT Datei/Pfad nicht gefunden

_doserrno wird auf einen der folgenden Werte gesetzt:

ENOENT Datei/Pfad nicht gefunden
ENMFILE Keine weiteren passenden Einträge

Siehe auch

findfirst

floor, floorl math.h

Funktion Rundet ab.

Syntax

```
double floor(double x);
long double floorl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
floor	■	■	■	■	■	■	■
floorl	■		■	■			■

Beschreibung *floor* rundet auf den größten Integerwert ab, der nicht größer als *x* ist. *floorl* ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert *floor* liefert den Integer als **double**-Wert zurück. *floorl* liefert den Integer als Wert vom Typ **long double** zurück.

Siehe auch *ceil*, *fmod*

flushall stdio.h

Funktion Entleert die Puffer aller Streams.

Syntax

```
int flushall(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *flushall* löscht alle zu offenen Eingabe-Streams gehörigen Puffer und schreibt alle zu offenen Ausgabe-Streams gehörigen Puffer in ihre jeweiligen Dateien. Die auf *flushall* folgenden Leseoperationen lesen aus ihren Eingabedateien neue Daten in die Puffer ein. Durch *flushall* werden keine Streams geschlossen.

Rückgabewert *flushall* liefert die Gesamtzahl der offenen Ein- und Ausgabe-Streams als Integerwert zurück.

Siehe auch *fclose*, *fcloseall*, *fflush*

_fmemccpy

Siehe *memccpy*.

F

_fmemchr

Siehe *memchr*.

_fmemcmp

Siehe *memcmp*.

_fmemcpy

Siehe *memcpy*.

_fmemicmp

Siehe *memicmp*.

_fmemmove

Siehe *memmove*.

_fmemset

Siehe *memset*.

fmod, fmodl math.h

Funktion Berechnet *x* Modulo *y*, den Rest der Division *x*/*y*.

Syntax

```
double fmod(double x, double y);
long double fmodl(long double x, long double y);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
fmod	■	■	■	■	■	■	■
fmodl	■		■	■			■

Beschreibung *fmod* berechnet *x* Modulo *y* (den Rest *f*, wobei $x = ay + f$ für einen Integer *a* und $0 <= f < y$ gilt). *fmodl* ist die **long double** Version. Sie erwartet Argumente vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert *fmod* und *fmodl* liefern den Rest *f*, wobei $x = ay + f$ gilt (wie beschrieben). Wenn *y* gleich 0 ist, liefern *fmod* und *fmodl* den Wert 0 zurück.

Siehe auch *ceil*, *floor*, *modf*

_fmovmem

Siehe *movmem*.

fnmerge — dir.h

F

Funktion Erzeugt einen Pfadnamen aus einzelnen Komponenten.

Syntax

```
void fnmerge(char *path, const char *drive, const char *dir, const char *name,
             const char *ext);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *fnmerge* erzeugt einen Pfadnamen aus Einzelkomponenten und speichert ihn in dem durch *path* bezeichneten String.

Der neue Pfadname heißt

```
X:\DIR\SUBDIR\NAME.EXT
```

wobei gilt:

drive	= X
dir	= \DIR\SUBDIR\
name	= NAME
ext	= .EXT

fnmerge setzt voraus, daß der mit *path* übergebene String eine ausreichende Größe für den konstruierten Pfadnamen hat. Die Maximalgröße ist durch die in dir.h definierte Konstante MAXPATH festgelegt.

fnmerge und *fnsplit* verhalten sich invers zueinander. Die Anwendung von *fnmerge* auf die durch *fnsplit* ermittelten Elemente eines vollständigen Pfadnamens ergeben wieder das Original.

Rückgabewert Keiner.

Siehe auch *fnsplit*

fnsplit dir.h

Funktion Zerlegt einen vollständigen Pfadnamen in Einzelkomponenten.

Syntax

```
int fnsplit(const char *path, char *drive, char *dir, char *name, char *ext);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *fnsplit* übernimmt einen vollständigen Dateinamen (Parameter *path*) als String in der Form

```
X:\DIR\SUBDIR\NAME.EXT
```

und zerlegt *path* in seine vier Einzelkomponenten. Diese werden dann in den durch *drive*, *dir*, *name* und *ext* bezeichneten Strings gespeichert. (Alle fünf Parameter müssen übergeben werden, einzelne davon dürfen aber NULL sein, was zur Folge hat, daß die zugehörige Komponente zwar analysiert, aber nicht gespeichert wird.)

Die Maximalgrößen aller beteiligen Strings (inklusive des abschließenden Nullzeichens \0) sind durch die Konstanten MAXDRIVE, MAXDIR, MAXPATH, MAXFILE und MAXEXT in dir.h festgelegt.

Konstante	Max. 16 Bit	Max. 32 Bit	String
MAXPATH	80	260	*path*
MAXDRIVE	3	3	*drive* inklusive Doppelpunkt (:)
MAXDIR	66	256	*dir* inklusive führendem und abschließendem umgekehrten Schrägstrich (\)
MAXFILE	9	256	*name*
MAXEXT	5	256	*ext* inklusive führendem Punkt (.)

fnsplit unterstellt, daß die durch *drive*, *dir*, *name* und *ext* bezeichneten Strings eine ausreichende Größe haben, um jede Komponente zu speichern, die nicht NULL ist.

fnsplit behandelt die Interpunktionszeichen aus *path* in folgender Weise:

- *drive* enthält den Laufwerksbezeichner mit einem Doppelpunkt (C:, A:, usw.).
- *dir* enthält den führenden und den abschließenden umgekehrten Schrägstrich (\BC4\INCLUDE\, \SOURCE\, usw.).
- *name* enthält den Dateinamen.
- *ext* enthält den Punkt, der die Namenserweiterung einleitet (.C, .EXE, usw.).

fnmerge und *fnsplit* verhalten sich invers zueinander. Die Anwendung von *fnmerge* auf die durch *fnsplit* ermittelten Elemente eines vollständigen Pfadnamens ergeben wieder das Original.

Rückgabewert

fnsplit liefert einen Integerwert zurück, der sich aus fünf in dir.h definierten Flags zusammensetzt. Ein gesetztes Flag zeigt an, daß die zugehörige Komponente in *path* vorhanden war. Folgende Flags sind in dir.h definiert:

EXTENSION	Eine Namenserweiterung
FILENAME	Ein Dateiname
DIRECTORY	Mindestens ein Verzeichnis
DRIVE	Ein Laufwerksbezeichner (siehe dir.h)
WILDCARDS	Jokerzeichen (* und/oder ?)

Siehe auch

fnmerge

fopen — stdio.h

F

Funktion

Öffnet einen Stream.

Syntax

```
FILE *fopen(const char *filename, const char *mode);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

fopen öffnet die durch *filename* bezeichnete Datei und ordnet ihr einen Stream zu. Zurückgeliefert wird ein Zeiger auf diesen Stream, über den weitere Operationen mit diesem Stream ausgeführt werden können.

Der Parameter *mode* kann einen der folgenden Werte annehmen:

Wert	Beschreibung
r	Öffnung ausschließlich für Leseoperationen.
w	Erzeugung einer Datei für Schreiboperationen. Falls die Datei bereits existiert, wird sie überschrieben.
a	Öffnung für Schreiboperationen. Falls die Datei bereits existiert, werden zusätzliche Informationen an das aktuelle Dateiende angefügt, ansonsten wird sie automatisch erzeugt.
r+	Öffnung einer Datei für Lese- und Schreiboperationen. Die Datei muß bereits existieren.
w+	Erzeugung einer neuen Datei für Lese- und Schreiboperationen. Falls die Datei bereits existiert, wird sie überschrieben.
a+	Öffnung einer Datei zum Lesen und Anhängen neuer Daten durch Schreiboperationen. Falls die Datei bereits existiert, bleibt ihr alter Inhalt erhalten – falls nicht, wird sie automatisch erzeugt.

Bei der Öffnung kann zusätzlich explizit angegeben werden, ob die Datei im Modus *text* (t) oder *binary* (b) bearbeitet werden soll. Die Angabe geschieht durch ein einfaches Anhängen des jeweiligen Buchstabens (also

wb, *w+b*, *rt*, *a+t* etc.). Im Parameter *mode* kann bei *fopen* das Zeichen *t* bzw. *b* auch zwischen den Buchstaben und dem Pluszeichen eingefügt werden. So ist beispielsweise *rt+* identisch mit *r+t*.

Wenn die Angabe *b* oder *t* im *mode*-String fehlt, wird der jeweilige Modus durch die globale Variable *_fmode* bestimmt, deren Wert über die direkte Zuweisung einer der in fcntl.h definierten Konstanten O_TEXT oder O_BINARY festgelegt werden kann.

Mit *r+* und *w+* erzeugte Dateien können sowohl gelesen als auch beschrieben werden – zwischen einem Wechsel von »Schreiben« nach »Lesen« muß allerdings ein Aufruf von *fseek* oder *rewind* stattfinden. Zwischen einem Wechsel von »Lesen« nach »Schreiben« muß ebenfalls *fseek* oder *rewind* stattfinden, es sei denn, die Leseoperation hat das Dateiende erreicht.

Rückgabewert *fopen* liefert bei fehlerfreier Ausführung einen Zeiger auf den neu geöffneten Stream, im Falle eines Fehlers den Wert NULL zurück.

Siehe auch *creat*, *dup*, *fclose*, *fdopen*, *ferror*, *_fmode* (globale Variable), *fread*, *freopen*, *fseek*, *fwrite*, *open*, *rewind*, *setbuf*, *setmode*

FP_OFF, FP_SEG — dos.h

Funktion Ermittelt den Offset- oder Segment-Anteil einer **far**-Adresse.

Syntax

```
unsigned FP_OFF(void far *p);
unsigned FP_SEG(void far *p);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *FP_OFF* ist ein Makro, das den Offset-Anteil des **far**-Zeigers *p* ermittelt oder setzt.

FP_SEG ist ein Makro, das den Segment-Anteil des **far**-Zeigers *p* ermittelt oder setzt.

Rückgabewert *FP_OFF* liefert den Offset-Anteil eines Zeigers als **unsigned int** zurück.

FP_SEG liefert den Segment-Anteil eines Zeigers als **unsigned int** zurück.

Siehe auch *MK_FP*, *movedata*, *segread*

_fpreset — float.h

Funktion Initialisiert die Gleitkommafunktionen neu.

Syntax

```
void _fpreset(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

_fpreset reinitialisiert die Gleitkommafunktionen von Borland C++ und wird normalerweise zusammen mit *system*, *exec...* oder *spawn...* verwendet. Die Funktion wird außerdem bei Gleitkommafehlern vor dem Aufruf von *longjmp* eingesetzt.

Wird ein 80x87 Koprozessor im Programm verwendet, so kann ein Kindprozeß (der von *system*, *exec...* oder *spawn...* gestartet wurde) den Gleitkommastatus seines Elternprozesses ändern.

Wenn ein 80x87 Koprozessor verwendet wird, sollten daher die folgenden Vorsichtsmaßnahmen getroffen werden:

- Die Funktionen *system*, *exec...* und *spawn...* dürfen nicht aufgerufen werden, während ein Gleitkommaausdruck ausgewertet wird.
- Nach Aufrufen von *system*, *exec...* und *spawn...* sollte der Gleitkommastatus mit *_fpreset* zurückgesetzt werden, wenn die Möglichkeit besteht, daß der Kindprozeß Gleitkommaoperationen mit dem 80x87 durchgeführt hat.

Rückgabewert

Keiner.

Siehe auch

_clear87, *_control87*, *_status87*

fprintf — stdio.h

Funktion

Schreibt formatiert in einen Stream.

Syntax

```
int fprintf(FILE *stream, const char *format[, argument, ...]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

fprintf übernimmt eine Folge von Argumenten, wendet auf jedes die im Formatstring (Parameter *format*) festgelegte Formatierungsanweisung an und schreibt die formatierten Daten in einen Stream. Die Anzahl der Formatierungsanweisungen und der Argumente muß übereinstimmen.

Weitere Details über Formatierungsanweisungen finden Sie Sie unter *printf.*

Rückgabewert

fprintf liefert bei fehlerfreier Ausführung die Anzahl der ausgegebenen Bytes, im Fehlerfall den Wert EOF zurück.

Siehe auch

cprintf, *fscanf*, *printf*, *putc*, *sprintf*

fputc — stdio.h

Funktion Schreibt ein Zeichen in einen Stream.

Syntax

```
int fputc(int c, FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *fputc* schreibt das Zeichen *c* in den durch *stream* angegebenen Stream.

In Win32 oder Win32s GUI-Anwendungen muß *stdout* umgelenkt werden.

Rückgabewert *fputc* liefert das übergebene Zeichen *c*, im Fehlerfall den Wert EOF zurück.

Siehe auch *fgetc*, *putc*

fputchar — stdio.h

Funktion Schreibt ein Zeichen in *stdout*.

Syntax

```
int fputchar(int c);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■	■		■

Beschreibung *fputchar* schreibt das Zeichen *c* in *stdout* und hat dieselbe Wirkung wie *fputc(c, stdout)*.

In Win32 oder Win32s GUI-Anwendungen muß *stdout* umgelenkt werden.

Rückgabewert *fputchar* liefert bei fehlerfreier Ausführung das übergebene Zeichen *c*, im Fehlerfall den Wert EOF zurück.

Siehe auch *fgetchar*, *freopen*, *putchar*

fputs — stdio.h

Funktion Schreibt einen String in einen Stream.

Syntax

```
int fputs(const char *s, FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *fputs* schreibt den nullterminierten String *s* in den angegebenen Ausgabe-Stream. Diese Funktion erzeugt *keinen* automatischen Zeilenvorschub am

Ende der Ausgabe und das abschließende Nullzeichen (\0) wird nicht ausgegeben.

Rückgabewert Bei fehlerfreier Ausführung liefert *fputs* das zuletzt geschriebene Zeichen (einen positiven Wert), im Falle eines Fehlers den Wert EOF zurück.

Siehe auch *fgets*, *gets*, *puts*

fread stdio.h

Funktion Liest Daten aus einem Stream.

Syntax

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

F

Beschreibung *fread* liest *n* Datenelemente, die jeweils die Größe *size* Bytes haben, aus dem angegebenen Stream in den Speicherbereich, auf den *ptr* zeigt.

Es werden insgesamt *(n * size)* Bytes gelesen.

Rückgabewert *fread* liefert bei fehlerfreier Ausführung die Anzahl der gelesenen Datenelemente (nicht Bytes) zurück. Im Falle eines Fehlers oder am Dateiende wird die Anzahl von Elementen (möglicherweise 0) zurückgeliefert, die noch fehlerfrei gelesen werden konnten.

Siehe auch *fopen*, *fwrite*, *printf*, *read*

free stdlib.h

Funktion Gibt einen reservierten Speicherblock wieder frei.

Syntax

```
void free(void *block);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *free* gibt einen zuvor mit *calloc*, *malloc* oder *realloc* reservierten Block wieder frei.

Rückgabewert Keiner.

Siehe auch *calloc*, *freemem*, *malloc*, *realloc*, *strdup*

freopen stdio.h

Funktion Ordnet einem offenen Stream eine neue Datei zu.

Syntax

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

freopen ersetzt die zu *stream* gehörende, offene Datei durch die über den Parameter *filename* angegebene Datei. Dabei wird – unabhängig vom Erfolg des anschließenden Öffnens – der Stream in jedem Fall geschlossen. Meistens wird diese Funktion verwendet, um *stdin*, *stdout* und *stderr* eine andere Datei zuzuordnen.

Der Parameterstring *mode* kann einen der folgenden Werte annehmen:

Wert	Beschreibung
r	Öffnung ausschließlich für Leseoperationen.
w	Erzeugung einer Datei für Schreiboperationen.
a	Öffnung für Schreiboperationen. Falls die Datei bereits existiert, werden zusätzliche Informationen an das aktuelle Dateiende angefügt, ansonsten wird sie automatisch erzeugt.
r+	Öffnung einer Datei für Lese- und Schreiboperationen. Die Datei muß bereits existieren.
w+	Erzeugung einer neuen Datei für Lese- und Schreiboperationen.
a+	Öffnung einer Datei zum Lesen und Anhängen neuer Daten durch Schreiboperationen. Falls die Datei bereits existiert, bleibt ihr alter Inhalt erhalten – falls nicht, wird sie automatisch erzeugt.

Bei der Öffnung kann zusätzlich explizit angegeben werden, ob die Datei im Modus *text* (t) oder *binary* (b) bearbeitet werden soll. Die Angabe geschieht durch ein einfaches Anhängen des jeweiligen Buchstabens (also *wb*, *w+b*, *rt*, *a+t* etc.). Wenn sie fehlt, wird der jeweilige Modus durch die globale Variable *_fmode* bestimmt, deren Wert über die direkte Zuweisung einer der in fcntl.h definierten Konstanten O_TEXT oder O_BINARY festgelegt werden kann.

Mit *r+* und *w+* erzeugte Dateien können sowohl gelesen, als auch beschrieben werden – zwischen einem Wechsel von »Schreiben« nach »Lesen« muß allerdings ein *fseek* oder *rewind* stattfinden. Zwischen einem Wechsel von »Lesen« nach »Schreiben« muß ebenfalls *fseek* oder *rewind* stattfinden, es sei denn, die Leseoperation hat das Dateiende erreicht.

Rückgabewert *freopen* liefert bei fehlerfreier Ausführung das Argument *stream*, im Fehlerfall den Wert NULL zurück.

Siehe auch *fclose, fdopen, fopen, open, setmode*

frexp, frexpl — math.h

Funktion Ermittelt die Mantisse und den Exponenten einer Gleitkommazahl.

Syntax

```
double frexp(double x, int *exponent);
long double frexpl(long double x, int *exponent);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
frexp	■	■	■	■	■	■	■
frexpl	■		■	■			■

Beschreibung *frexp* berechnet zu *x* die Mantisse *m* (ein **double**-Wert, dessen absoluter Betrag größer/gleich 0.5 und kleiner als 1.0 ist) und den Integerwert *n* so, daß die Gleichung $x = m * 2^n$ gilt. Der Exponent *n* wird in dem Integer gespeichert, auf den der Parameter *exponent* zeigt.

frexpl ist die **long double** Version. Sie erwartet für *x* ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert *frexp* und *frexpl* liefern die Mantisse *m* zurück. Die Fehlerbehandlung für *frexp* und *frexpl* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

Siehe auch *exp, ldexp, _matherr*

fscanf — stdio.h

Funktion Liest und formatiert Eingaben aus einem Stream.

Syntax

```
int fscanf(FILE *stream, const char *format[, address, ...]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *fscanf* liest eine Reihe von Eingabefeldern zeichenweise aus einem Stream. Entsprechend den Formatierungsanweisungen, die *fscanf* im Formatstring *format* übergeben wurden, wird jedes Feld gelesen, formatiert und an einer Adresse gespeichert, die nach *format* übergeben wurde. Die Anzahl der Formatierungsanweisungen, der übergebenen Adressen und der Eingabefelder müssen übereinstimmen.

Unter der Funktion *scanf* finden Sie eine ausführliche Beschreibung der Formatierungsanweisungen.

fscanf kann das Lesen eines bestimmten Eingabefeldes abbrechen, bevor das normale Feldende (ein Whitespace-Zeichen) erreicht ist. *fscanf* kann auch aus verschiedenen Gründen den Lesevorgang vollständig abbrechen. Die möglichen Ursachen hierfür werden unter *scanf* erläutert.

Rückgabewert

fscanf liefert die Anzahl der fehlerfrei gelesenen, konvertierten und gespeicherten Felder zurück. Gelesene, aber nicht gespeicherte Felder werden nicht mitgezählt.

Der Versuch, über das Ende einer Datei hinauszulesen, ergibt den Wert EOF. Wenn überhaupt keine Felder gespeichert wurden, wird 0 zurückgeliefert.

Siehe auch

atof, *cscanf*, *fprintf*, *printf*, *scanf*, *sscanf*, *vfscanf*, *vscanf*, *vsscanf*

fseek — stdio.h

Funktion

Setzt den Dateizeiger eines Streams auf eine neue Position.

Syntax

```
int fseek(FILE *stream, long offset, int whence);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

fseek setzt den zu *stream* gehörenden Dateizeiger auf die neue Position, die *offset* Bytes von der durch *whence* angegebenen Dateiposition entfernt ist. Für Streams im Textmodus sollte *offset* 0 sein oder ein Wert, der von *ftell* geliefert wurde.

Die Art der Positionierung wird über drei in stdio.h definierte Konstanten für *whence* festgelegt:

Konstante	*whence*	Dateiposition
SEEK_SET	(Wert 0)	Relativ zum Dateianfang
SEEK_CUR	(Wert 1)	Relativ zur aktuellen Position
SEEK_END	(Wert 2)	Relativ zum Dateiende

Mit *ungetc* zurückgestellte Zeichen sind nach *fseek* nicht mehr erreichbar. Benutzen Sie *fseek* für Stream-Ein-/Ausgabe und *lseek* für Datei-Handle-Ein-/Ausgabe.

In Dateien, die für Lese- *und* Schreiboperationen geöffnet worden sind, kann nach *fseek* zwischen »Lesen« und »Schreiben« gewechselt werden.

Rückgabewert

fseek liefert bei fehlerfreier Ausführung den Wert 0 zurück, ansonsten einen Wert ungleich null.

Es ist möglich, daß *fseek* 0 zurückliefert, obwohl bei der Ausführung der Funktion ein Fehler aufgetreten ist. Dies liegt daran, daß der Zeiger letztendlich von DOS gesetzt und das Ergebnis der Positionierung nicht überprüft wird. *fseek* gibt nur für ungeöffnete Dateien oder Geräte einen Fehlercode zurück.

Im Fehlerfall wird die globale Variable *errno* auf einen der folgenden Werte gesetzt:

EBADF	Ungültiger Dateizeiger
EINVAL	Ungültiges Argument
ESPIPE	Suche auf diesem Gerät nicht möglich

Siehe auch *fgetpos*, *fopen*, *fsetpos*, *ftell*, *lseek*, *rewind*, *setbuf*, *tell*

fsetpos — stdio.h

Funktion Positioniert den Dateizeiger eines Stream.

Syntax

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *fsetpos* setzt den Dateizeiger von *stream* auf eine neue Position. Die neue Position ist der Wert, der durch einen vorherigen Aufruf von *fgetpos* für diesen Stream ermittelt wurde. Das zum Stream gehörende Dateiende-Flag wird gelöscht; zuvor mit *ungetc* zurückgestellte Zeichen sind nach dieser Operation nicht mehr verfügbar.

Die nächste Operation nach einem Aufruf von *fsetpos* kann sowohl eine Lese- als auch eine Schreiboperation sein.

Rückgabewert *fsetpos* liefert bei fehlerfreier Ausführung den Wert 0 zurück. Im Fehlerfall wird ein Wert ungleich null zurückgeliefert und die globale Variable *errno* wird auf einen Wert ungleich null gesetzt.

Siehe auch *fgetpos*, *fseek*, *ftell*

_fsopen — stdio.h, share.h

Funktion Öffnet einen Stream für gemeinsamen Zugriff.

Syntax

```
FILE *_fsopen(const char *filename, const char *mode, int shflag);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

_fsopen öffnet die durch *filename* angegebene Datei und ordnet ihr einen Stream zu. *_fsopen* liefert einen Zeiger zurück, über den bei anschließenden Operationen auf den Stream zugegriffen werden kann.

Der Parameterstring *mode* kann einen der folgenden Werte annehmen:

Wert	Beschreibung
r	Öffnung ausschließlich für Leseoperationen.
w	Erzeugung einer Datei für Schreiboperationen.
a	Öffnung für Schreiboperationen. Falls die Datei bereits existiert, werden zusätzliche Informationen an das aktuelle Dateiende angefügt, ansonsten wird sie automatisch erzeugt.
r+	Öffnung einer Datei für Lese- und Schreiboperationen. Die Datei muß bereits existieren.
w+	Erzeugung einer neuen Datei für Lese- und Schreiboperationen. Existiert bereits eine Datei des angegebenen Namens, wird sie überschrieben.
a+	Öffnung einer Datei zum Lesen und Anhängen neuer Daten durch Schreiboperationen. Falls die Datei bereits existiert, bleibt ihr alter Inhalt erhalten – falls nicht, wird sie automatisch erzeugt.

Bei der Öffnung kann zusätzlich explizit angegeben werden, ob die Datei im Modus *text* (t) oder *binary* (b) bearbeitet werden soll. Die Angabe geschieht durch ein einfaches Anhängen des jeweiligen Buchstabens (also *wb*, *w+b*, *rt*, *a+t* etc.). Wenn sie fehlt, wird der jeweilige Modus durch die globale Variable *_fmode* bestimmt, deren Wert über die direkte Zuweisung einer der in fcntl.h definierten Konstanten O_TEXT oder O_BINARY festgelegt werden kann.

Mit *r+* und *w+* erzeugte Dateien können sowohl gelesen als auch beschrieben werden – zwischen einem Wechsel von »Schreiben« nach »Lesen« muß allerdings ein *fseek* oder *rewind* stattfinden. Zwischen einem Wechsel von »Lesen« nach »Schreiben« muß ebenfalls *fseek* oder *rewind* stattfinden, es sei denn, die Leseoperation hat das Dateiende erreicht.

shflag legt den Typ des gemeinsamen Zugriffs fest, der für die Datei *filename* zugelassen werden soll. Der DOS-Befehl SHARE muß ausgeführt worden sein, damit *shflag* nicht ignoriert wird. Die symbolischen Konstanten für *shflag* sind in share.h definiert:

Wert von shflag	Beschreibung
SH_COMPAT	Setzt Kompatibilitätsmodus
SH_DENYRW	Lesen und Schreiben nicht erlaubt
SH_DENYWR	Schreiben nicht erlaubt
SH_DENYRD	Lesen nicht erlaubt
SH_DENYNONE	Lesen und Schreiben erlaubt
SH_DENYNO	Lesen und Schreiben erlaubt

Rückgabewert

_fsopen liefert bei fehlerfreier Ausführung einen Zeiger auf den neu geöffneten Stream, im Fehlerfall NULL zurück.

Siehe auch

creat, *_dos_open*, *dup*, *fclose*, *fdopen*, *ferror*, *_fmode* (globale Variable), *fopen*, *fread*, *freopen*, *fseek*, *fwrite*, *open*, *rewind*, *setbuf*, *setmode*, *sopen*

fstat, stat — sys\stat.h

Funktion

Liefert Informationen über eine offene Datei.

Syntax

```
int fstat(int handle, struct stat *statbuf);
int stat(char *path, struct stat *statbuf);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

fstat speichert in der Struktur des Typs *stat* Informationen über die offene Datei oder das Verzeichnis, die bzw. das mit *handle* angegeben ist.

Die Funktion *stat* speichert Informationen über die angegebene Datei oder das Verzeichnis in einer Struktur des Typs *stat*. Der Dateiname wird durch den Parameter *path* angegeben.

statbuf zeigt auf eine *stat*-Struktur, die in sys\stat.h definiert ist und folgende Felder enthält:

st_mode	Bitmuster mit Informationen über den Modus der Datei
st_dev	Nummer des Laufwerks, das die Datei enthält, oder ein Datei-Handle, wenn die Datei auf einem Gerät ist
st_rdev	wie *st_dev*.
st_nlink	immer eine Integerkonstante mit dem Wert 1
st_size	Größe der Datei in Bytes
st_atime	Datum/Uhrzeit der letzten Änderung
st_mtime	wie *st_atime*
st_ctime	wie *st_atime*

Der Strukturtyp *stat* enthält noch drei weitere Felder, die hier nicht aufgeführt sind, da die darin enthaltenen Werte nur unter UNIX von Bedeutung sind.

Das Bitmuster *st_mode* setzt sich aus mehreren Einzelinformationen zusammen, die in sys\stat.h als Bit-Konstanten definiert sind.

Eines dieser beiden Bits ist gesetzt:

S_IFCHR Wenn *handle* sich auf ein Gerät bezieht.
S_IFREG Wenn es sich um eine reguläre Datei handelt.

Eines oder beide der folgenden Bits sind gesetzt:

S_IWRITE Wenn der Benutzer Schreiberlaubnis hat.
S_IREAD Wenn der Benutzer Leseerlaubnis hat.

Die Dateisysteme HPFS und NTFS machen zusätzlich folgende Unterschiede:

st_atime Letzter Zugriff
st_mtime Letzte Änderung
st_ctime Uhrzeit der Dateierzeugung

Rückgabewert

fstat und *stat* liefern den Wert 0 zurück, wenn sie die Informationen über die offene Datei erfolgreich ermitteln konnten. Im Fehlerfall ist der Rückgabewert -1 und die globale Variable *errno* wird auf folgenden Wert gesetzt:

EBADF Ungültiges Handle

Siehe auch

access, *chmod*

_fstr* string.h

Funktion

Stringfunktionen im Speichermodell large.

Syntax

`far-Stringfunktionen`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Ist der Rückgabewert einer **far**-Stringfunktion vom Typ **int** oder *size_t*, wird er nicht vom Schlüsselwort **far** modifiziert.

Im Abschnitt **Siehe auch** weiter unten finden Sie eine Liste der String-Funktionen, zu denen es eine **far**-Version gibt. Der **far**-Version einer String-Funktion wird die Namenserweiterung *_fstr* vorangestellt. Die **far**-Versionen verhalten sich genau wie die entsprechenden Standardversionen, mit der einen Ausnahme, daß Parameter und Rückgabewert (sofern er vom Typ **char far *** ist) durch das Schlüsselwort **far** modifiziert

werden. In der Beschreibung der Standardfunktion ist die jeweilige **far**-Version enthalten.

Rückgabewert Ist der Rückgabewert der Standardfunktion ein **char**-Zeiger, so gibt die *_fstr*-Version einen **char far**-Zeiger zurück.

Siehe auch *strcat, strchr, strcmp, strcpy, strcspn, strdup, stricmp, strlen, strlwr, strncat, strncmp, strnicmp, strnset, strpbrk, strrchr, strrev, strset, strspn, strstr, strtok, strupr*

ftell stdio.h

Funktion Liefert den aktuellen Dateizeiger.

Syntax

```
long int ftell(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *ftell* liefert den aktuellen Dateizeiger für *stream* zurück. Die Position wird relativ zum Dateianfang in Bytes gemessen (falls es sich um eine Binärdatei handelt). Der von *ftell* zurückgelieferte Wert kann bei einem nachfolgenden Aufruf von *fseek* verwendet werden.

Rückgabewert *ftell* liefert bei fehlerfreier Ausführung die aktuelle Position des Dateizeigers zurück. Im Fehlerfall ist der Rückgabewert -1L und die globale Variable *errno* wird auf einen der beiden folgenden Werte gesetzt:

EBADF	Ungültiger Dateizeiger
ESPIPE	Suche auf diesem Gerät nicht möglich

Siehe auch *fgetpos, fseek, fsetpos, lseek, rewind, tell*

ftime sys\timeb.h

Funktion Speichert die Systemzeit in einer Struktur des Typs *timeb*.

Syntax

```
void ftime(struct timeb *buf);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung Unter UNIX ist *ftime* nur unter System V verfügbar.

ftime ermittelt die aktuelle Systemzeit und speichert sie in den Feldern der durch *buf* bezeichneten *timeb*-Struktur. Der Strukturtyp *timeb* enthält die vier Felder *time*, *millitm*, *_timezone* und *dstflag*:

```
struct timeb {
    long  time;
    short millitm;
    short _timezone;
    short dstflag;
};
```

- *time* ist die Zeit in Sekunden seit dem 1.1.1970 00:00:00 GMT (Greenwich Mean Time).
- *millitm* ist der Bruchteil einer Sekunde in Millisekunden
- *_timezone* ist der lokale Zeitunterschied zur GMT in Minuten. Der Zeitunterschied zur GMT wird über die globale Variable *_timezone* berechnet, deren Wert durch *tzset* gesetzt wird.
- *dstflag* gibt an, ob in der aktuellen Zeitzone eine Verschiebung durch die Sommerzeit berücksichtigt wird.

Da *ftime* die Funktion *tzset* aufruft, ist es nicht nötig, vorher *tzset* explizit aufzurufen.

Rückgabewert Keiner.

Siehe auch *asctime*, *ctime*, *gmtime*, *localtime*, *stime*, *time*, *tzset*

_fullpath — stdlib.h

Funktion Konvertiert einen relativen Pfadnamen in einen absoluten.

Syntax

```
char * _fullpath(char *buffer, const char *path, int buflen);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_fullpath* konvertiert den relativen Pfadnamen in *path* in einen absoluten Pfadnamen. Dieser wird in dem Zeichen-Array gespeichert, auf das *buffer* zeigt. *buflen* gibt an, wieviele Zeichen in *buffer* gespeichert werden können. *_fullpath* liefert NULL zurück, wenn der Puffer nicht groß genug für den absoluten Pfadnamen ist oder der Pfad eine ungültige Laufwerksangabe enthält.

Wenn *buffer* NULL ist, reserviert *_fullpath* einen Puffer mit bis zu _MAX_PATH Zeichen. Dieser Puffer sollte durch einen Aufruf von *free* freigegeben werden, wenn er nicht mehr benötigt wird. Die Konstante _MAX_PATH ist in stdlib.h definiert.

Rückgabewert Bei fehlerfreier Ausführung liefert *_fullpath* einen Zeiger auf den Puffer zurück, der den absoluten Pfadnamen enthält. Andernfalls ist das Ergebnis NULL.

Siehe auch *_makepath*, *_splitpath*

fwrite — stdio.h

Funktion Schreibt Daten in einen Stream.

Syntax

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *fwrite* fügt *n* Datenelemente der Größe *size* Bytes an die durch *stream* gegebene Datei an. Die zu schreibenden Daten beginnen bei der Adresse, auf die *ptr* zeigt. Es werden *n * size* Bytes geschrieben. *ptr* kann auf beliebige Objekte zeigen.

Rückgabewert *fwrite* liefert bei fehlerfreier Ausführung die Anzahl der geschriebenen Datenelemente (nicht Bytes) zurück. Im Fehlerfall wird die Anzahl der Datenelemente zurückgeliefert, die noch fehlerfrei geschrieben werden konnten.

Siehe auch *fopen*, *fread*

gcvt — stdlib.h

Funktion Konvertiert eine Gleitkommazahl in einen String.

Syntax

```
char *gcvt(double value, int ndec, char *buf);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *gcvt* konvertiert *value* in einen nullterminierten ASCII-String, der in *buf* gespeichert wird. Die Funktion versucht, *ndec* signifikante Ziffern im F-Format von FORTRAN zu erzeugen. Wenn dies nicht möglich ist, liefert sie den Wert im E-Format von *printf* (druckfertig formatiert). Abschließende Nullen werden möglicherweise unterdrückt.

Rückgabewert *gcvt* liefert die Adresse des durch *buf* bezeichneten Strings zurück.

Siehe auch *ecvt*, *fcvt*, *sprintf*

geninterrupt — dos.h

Funktion Erzeugt einen Software-Interrupt.

Syntax `void geninterrupt(int intr_num);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung Das Makro *geninterrupt* löst einen Software-Interrupt für den Interrupt *intr_num* aus. Der Zustand der Register nach dem Aufruf hängt von dem aufgerufenen Interrupt ab.

Interrupts können Register in einem nicht vorhersehbaren Zustand zurücklassen.

Rückgabewert Keiner.

Siehe auch *bdos*, *bdosptr*, *disable*, *enable*, *getvect*, *int86*, *int86x*, *intdos*, *intdosx*, *intr*

getc — stdio.h

Funktion Liest ein Zeichen aus einem Stream.

Syntax `int getc(FILE *stream);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *getc* ist ein Makro, das das jeweils nächste Zeichen aus dem angegebenen Eingabe-Stream liest, zurückliefert und den zugehörigen Dateizeiger entsprechend inkrementiert.

Rückgabewert *getc* liefert bei fehlerfreier Ausführung das gelesene Zeichen als **int** ohne Vorzeichenerweiterung zurück. Am Dateiende oder im Fehlerfall liefert die Funktion EOF zurück.

Siehe auch *fgetc*, *getch*, *getchar*, *getche*, *gets*, *putc*, *putchar*, *ungetc*

getcbrk — dos.h

Funktion Ermittelt, ob DOS auf [*Strg*][*Untbr*] prüft.

Syntax `int getcbrk(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *getcbrk* verwendet den DOS-Systemaufruf 0x33, um zu ermitteln, wann DOS auf [*Strg*][*Untbr*] prüft.

Rückgabewert *getcbrk* liefert 0, wenn die Überprüfung abgeschaltet ist, ansonsten den Wert 1 zurück.

Siehe auch *ctrlbrk*, *setcbrk*

getch conio.h

Funktion Liest ein Zeichen von der Tastatur, ohne es auf dem Bildschirm auszugeben.

Syntax

```
int getch(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *getch* liest ein einzelnes Zeichen von der Tastatur, ohne es als »Echo« auf dem Bildschirm auszugeben.

➥ Diese Funktion sollte in Win32 oder Win32s GUI-Anwendungen nicht verwendet werden.

Rückgabewert *getch* liefert das von der Tastatur gelesene Zeichen zurück.

Siehe auch *cgets*, *cscanf*, *fgetc*, *getc*, *getchar*, *getche*, *getpass*, *kbhit*, *putch*, *ungetch*

getchar stdio.h

Funktion Liest ein Zeichen aus *stdin*.

Syntax

```
int getchar(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■	■	■	■

Beschreibung *getchar* ist ein Makro, das das jeweils nächste Zeichen aus dem Eingabe-Stream *stdin* liest und zurückliefert. Es ist definiert als *getc*(*stdin*).

➥ In Win32s oder Win32 GUI-Anwendungen muß *stdin* umgelenkt werden.

Rückgabewert *getchar* liefert bei fehlerfreier Ausführung das gelesene Zeichen als **int**-Wert ohne Vorzeichenerweiterung zurück. Am Dateiende oder im Fehlerfall liefert die Funktion EOF zurück.

Siehe auch *fgetc*, *fgetchar*, *freopen*, *getc*, *getch*, *getche*, *gets*, *putc*, *putchar*, *scanf*, *ungetc*

getche — conio.h

Funktion Liest ein Zeichen von der Tastatur und gibt es auf dem Bildschirm aus.

Syntax

```
int getche(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *getche* liest ein einzelnes Zeichen von der Tastatur und gibt dieses Zeichen auf dem Bildschirm in das aktuelle Textfenster aus. *getche* schreibt direkt in den Bildschirmspeicher oder benutzt BIOS-Routinen.

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert *getche* liefert das von der Tastatur gelesene Zeichen zurück.

Siehe auch *cgets*, *cscanf*, *fgetc*, *getc*, *getch*, *getchar*, *kbhit*, *putch*, *ungetch*

getcurdir — dir.h

Funktion Ermittelt das aktuelle Verzeichnis eines Laufwerks.

Syntax

```
int getcurdir(int drive, char *directory);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *getcurdir* ermittelt das aktuelle Verzeichnis auf dem durch *drive* angegebenen Laufwerk. Wenn *drive* den Wert 0 hat, wird das aktuelle Laufwerk verwendet, ansonsten gilt A = 1, B = 2 usw. *directory* zeigt auf einen Speicherbereich der Größe MAXDIR – hier trägt *getcurdir* den nullterminierten Namen des ermittelten Verzeichnisses ein (ohne führenden Backslash und ohne Laufwerksbezeichner).

Rückgabewert Bei fehlerfreier Ausführung liefert *getcurdir* den Wert 0, im Fehlerfall -1 zurück.

Siehe auch *chdir*, *getcwd*, *getdisk*, *mkdir*, *rmdir*

getcwd — dir.h

Funktion Ermittelt das aktuelle Arbeitsverzeichnis.

Syntax `char *getcwd(char *buf, int buflen);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *getcwd* speichert den vollständigen Pfadnamen des aktuellen Arbeitsverzeichnisses (inklusive des Laufwerks) als String an der durch *buf* bezeichneten Adresse. Die Maximallänge des Pfadnamens wird durch den Parameter *buflen* begrenzt. Ist der vollständige Pfadname, einschließlich des abschließenden Nullzeichens, länger als *buflen* Bytes, so wird ein Fehler erzeugt.

Wenn der als *buf* übergebene Zeiger den Wert NULL hat, reserviert *getcwd* über einen Aufruf von *malloc* selbständig einen Bereich von *buflen* Bytes. Später können Sie diesen reservierte Speicherbereich wieder freigegeben, indem Sie der Funktion *free* den Rückgabewert von *getcwd* übergeben.

Rückgabewert *getcwd* liefert folgende Werte zurück:

- Ist *buf* bei der Eingabe ungleich NULL, wird bei fehlerfreier Ausführung *buf*, im Fehlerfall NULL zurückgeliefert.
- Ist *buf* bei der Eingabe gleich NULL, liefert *getcwd* bei fehlerfreier Ausführung einen Zeiger auf den reservierten Puffer zurück.

Im Fehlerfall wird *errno* auf einen der folgenden Werte gesetzt:

ENODEV	Gerät nicht bereit/nicht vorhanden
ENOMEM	Nicht genug Platz im Hauptspeicher (*buf* gleich NULL)
ERANGE	Verzeichnisname länger als *buflen* (*buf* ungleich NULL)

Siehe auch *chdir*, *getcurdir*, *_getdcwd*, *getdisk*, *mkdir*, *rmdir*

getdate

Siehe *_dos_getdate*.

_getdcwd — direct.h

Funktion Liefert das aktuelle Arbeitsverzeichnis des angegebenen Laufwerks.

Syntax `char *_getdcwd(int drive, char *buffer, int buflen);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

G

Beschreibung

_getdcwd liefert den vollständigen Pfadnamen des Arbeitsverzeichnisses im angegebenen Laufwerk (einschließlich des Laufwerknamens) als String in *buffer* zurück, dessen Maximallänge durch den Parameter *buflen* gesetzt wird. Ist der vollständige Pfadname einschließlich des abschließenden Nullzeichens länger als *buflen* Bytes, so wird ein Fehler erzeugt. Für den Parameter *drive* gilt: 0 = aktuelles Laufwerk, 1 = A, 2 = B, etc.

Wenn der als *buffer* übergebene Zeiger den Wert NULL hat, reserviert *_getdcwd* einen Puffer von mindestens *buflen* Bytes. Später kann dieser Speicherbereich durch Übergabe des Rückgabewerts von *_getdcwd* an die Funktion *free* wieder freigegeben werden.

Rückgabewert

_getdcwd liefert bei fehlerfreier Ausführung einen Zeiger auf den Puffer zurück, der den Namen des aktuellen Verzeichnisses im angegebenen Laufwerk enthält. Im Fehlerfall liefert *_getdcwd* den Wert NULL, und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

ENOMEM Nicht genug Platz im Hauptspeicher (*buffer* ist NULL)

ERANGE Verzeichnisname ist länger als *buflen* (*buffer* ist ungleich NULL)

Siehe auch

chdir, *getcwd*, *mkdir*, *rmdir*

getdfree — dos.h

Funktion

Ermittelt den freien Platz auf einer Diskette/Festplatte.

Syntax

```
void getdfree(unsigned char drive, struct dfree *dtable);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

getdfree ermittelt Informationen über das durch den Parameter *drive* angegebene Laufwerk und speichert sie in einer Struktur des Typs *dfree*. Wenn *drive* den Wert 0 hat, wird das aktuelle Laufwerk bearbeitet, ansonsten gilt A = 1, B = 2 usw.

Der Strukturtyp *dfree* ist folgendermaßen definiert:

```
struct dfree {
   unsigned df_avail;  /* freie Cluster */
   unsigned df_total;  /* Gesamtzahl Cluster */
   unsigned df_bsec;   /* Bytes pro Sektor */
   unsigned df_sclus;  /* Sektoren pro Cluster */
};
```

Rückgabewert *getdfree* liefert keinen Wert zurück. Im Fehlerfall wird das Element *df_sclus* auf (**unsigned**)-1 gesetzt.

Siehe auch *getfat*, *getfatd*

getdisk, setdisk — dir.h

Funktion Ermittelt oder setzt die aktuelle Laufwerksnummer.

Syntax

```
int getdisk(void);
int setdisk(int drive);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *getdisk* liefert die Nummer des aktuellen Laufwerks als Integer zurück: 0 entspricht Laufwerk A, 1 entspricht Laufwerk B usw. *setdisk* setzt das aktuelle Laufwerk auf die Nummer in *drive*: 0 entspricht Laufwerk A, 1 entspricht Laufwerk B usw.

Rückgabewert *getdisk* liefert die aktuelle Laufwerksnummer zurück. *setdisk* liefert die Anzahl der verfügbaren Laufwerke zurück.

Siehe auch *getcurdir*, *getcwd*

getdta — dos.h

Funktion Liefert die Adresse der aktuellen DTA zurück.

Syntax

```
char far *getdta(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *getdta* liefert die aktuelle *disk transfer address* (DTA).

In den Speichermodellen *Small* und *Medium* geht Borland C++ davon aus, daß sich die DTA im aktuellen Datensegment befindet. Solange ausschließlich mit C oder C++ gearbeitet wird, ist das auch der Fall – von Routinen in Maschinensprache kann die DTA aber auf eine beliebige Adresse gesetzt werden.

In den Speichermodellen Compact und Large liefert *getdta* die korrekte Adresse zurück, die sich auch außerhalb der Programmsegmente befinden kann.

Rückgabewert *getdta* liefert einen Zeiger auf die aktuelle DTA zurück.

Siehe auch *setdta*

G

getenv — stdlib.h

Funktion Ermittelt den Wert einer Umgebungsvariable.

Syntax

```
char *getenv(const char *name);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *getenv* liefert den Wert der durch den Parameter *name* angegebenen Umgebungsvariable zurück. Unter DOS und OS/2 muß *name* in Großbuchstaben vorliegen, in anderen Systemen kann er entweder in Groß- oder Kleinbuchstaben geschrieben sein. *name* darf das Gleichheitszeichen (=) nicht enthalten. Existiert die angegebene Umgebungsvariable nicht, so liefert *getenv* einen NULL-Zeiger zurück.

Über den Aufruf *getenv("name=")* kann eine Umgebungsvariable gelöscht werden.

Umgebungsvariablen dürfen nicht direkt geändert werden. Wenn ein bestimmter Wert geändert werden soll, muß *putenv* verwendet werden.

Rückgabewert Bei fehlerfreier Ausführung liefert *getenv* den Wert zurück, der der Variablen zugeordnet ist. Falls die angegebene Variable *name* nicht existiert, so liefert *getenv* einen NULL-Zeiger zurück.

Siehe auch *_environ* (globale Variable), *getpsp*, *putenv*

getfat — dos.h

Funktion Ermittelt Informationen über die FAT des angegebenen Laufwerks.

Syntax

```
void getfat(unsigned char drive, struct fatinfo *dtable);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *getfat* füllt die mit *dtable* angegebene Struktur des Typs *fatinfo* mit Informationen über die FAT (File Allocation Table) des Laufwerks *drive*. Wenn drive den Wert 0 hat, wird das aktuelle Laufwerk benutzt, ansonsten gilt A = 1, B = 2 usw. Der Strukturtyp *fatinfo* ist folgendermaßen definiert:

```
struct fatinfo {
   char fi_sclus;       /* Sektoren pro Cluster */
   char fi_fatid;       /* FAT ID-Byte */
   unsigned fi_nclus;   /* Gesamtzahl Cluster */
   int fi_bysec;        /* Bytes pro Sektor */
};
```

Rückgabewert Keiner.

Siehe auch *getdfree*, *getfatd*

getfatd dos.h

Funktion Ermittelt Informationen über die FAT des aktuellen Laufwerks.

Syntax

```
void getfatd(struct fatinfo *dtable);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *getfatd* füllt die über *dtable* angegebene Struktur des Typs *fatinfo* mit Informationen über die FAT des Datenträgers im aktuellen Laufwerk. Der Strukturtyp *fatinfo* ist folgendermaßen definiert:

```
struct fatinfo {
   char fi_sclus;    /* Sektoren pro Cluster */
   char fi_fatid;    /* FAT ID-Byte */
   int fi_nclus;     /* Gesamtzahl Cluster */
   int fi_bysec;     /* Bytes pro Sektor */
};
```

Rückgabewert Keiner.

Siehe auch *getdfree*, *getfat*

getftime, setftime io.h

Funktion Liest und setzt Datum und Uhrzeit einer Datei.

Syntax

```
int getftime(int handle, struct ftime *ftimep);
int setftime(int handle, struct ftime *ftimep);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *getftime* ermittelt Datum und Uhrzeit der durch *handle* bezeichneten, offenen Datei und speichert die entsprechenden Werte in der durch *ftimep* angegebenen *ftime*-Struktur.

setftime setzt Datum und Uhrzeit der durch *handle* bezeichneten, offenen Datei auf die Werte der *ftime*-Struktur, auf die *ftimep* zeigt. Nach dem Aufruf von *setftime* darf nicht in die Datei geschrieben werden, da die geänderten Daten ansonsten verloren gehen. Die Datei muß für Schreib-

zugriffe geöffnet worden sein, da ansonsten der Fehler EACCES auftritt.

Der Strukturtyp *ftime* ist folgendermaßen definiert:

```
struct ftime {
   unsigned ft_tsec: 5;     /* Zwei Sekunden */
   unsigned ft_min: 6;      /* Minuten */
   unsigned ft_hour: 5;     /* Stunden */
   unsigned ft_day: 5;      /* Tage */
   unsigned ft_month: 4;    /* Monate */
   unsigned ft_year: 7;     /* Jahr - 1980 */
 };
```

Rückgabewert *getftime* und *setftime* liefern bei fehlerfreier Ausführung den Wert 0.

Im Fehlerfall wird der Wert -1 zurückgeliefert und *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht erlaubt
EBADF	Ungültiges Handle
EINVFNC	Ungültige Funktionsnummer

Siehe auch *fflush, open, setftime*

getpass conio.h

Funktion Liest ein Paßwort.

Syntax `char *getpass(const char *prompt);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■			■

Beschreibung *getpass* gibt den durch *prompt* bezeichneten nullterminierten String aus und erwartet die Eingabe eines Paßworts über die Tastatur. Die vom Benutzer eingegebenen Zeichen werden *nicht* auf dem Bildschirm dargestellt. Der String hat eine Maximallänge von acht Zeichen (wobei das abschließende Nullzeichen nicht mitgezählt wird).

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert *getpass* liefert einen Zeiger auf einen statischen String zurück, der durch jeden erneuten Aufruf überschrieben wird.

Siehe auch *getch*

_get_osfhandle — io.h

Funktion Verbindet Datei-Handles.

Syntax `long _get_osfhandle(int filehandle);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			

Beschreibung *_get_osfhandle* verbindet ein Datei-Handle des Betriebssystems mit einem existierenden Laufzeit-Datei-Handle. Der Parameter *filehandle* ist ein Datei-Handle Ihres Programms.

Rückgabewert Bei erfolgreicher Ausführung liefert *_get_osfhandle* ein Datei-Handle des Betriebssystems zurück, das *filehandle* entspricht. Andernfalls wird der Wert -1 zurückgeliefert und *errno* wird auf EBADF (ungültiges Handle) gesetzt

Siehe auch *fdopen, filno, _open_osfhandle*

getpid — process.h

Funktion Liefert die Prozeß-ID eines Programms.

Syntax `unsigned getpid(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung Diese Funktion liefert die aktuelle Prozeß-ID zurück. Die Prozeß-ID ist ein Integerwert, der einen Prozeß eindeutig kennzeichnet.

Rückgabewert *getpid* liefert die Prozeß-ID des aktuellen Prozesses zurück.

Siehe auch *getpsp, _psp* (globale Variable)

getpsp — dos.h

Funktion Liefert die Adresse des Programmsegment-Präfix (PSP).

Syntax `unsigned getpsp(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *getpsp* verwendet den DOS-Funktionsaufruf 0x62, um die Segmentadresse des Programmsegment-Präfix (PSP) zu ermitteln.

Rückgabewert *getpsp* liefert die Segmentadresse des PSP zurück.

Siehe auch *getenv*, *_psp* (globale Variable),

gets — stdio.h

Funktion Liest einen String aus *stdin*.

Syntax

```
char *gets(char *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■	■	■	■

Beschreibung *gets* liest einen String aus dem Standardeingabe-Stream *stdin* und speichert ihn in der durch *s* angegebenen Stringvariablen. Die Eingabe wird durch einen Zeilenvorschub beendet, der in *s* als Nullzeichen (\0) gespeichert wird.

Eingabestrings dürfen bestimmte Whitespace-Zeichen (Leerzeichen, Tabs) enthalten. *gets* kehrt zurück, wenn ein Zeilenvorschub gelesen wird; der String bis zu diesem Zeilenvorschub wird in *s* gespeichert.

➡ In Win32 oder Win32s GUI-Anwendungen muß *stdin* umgelenkt werden.

Rückgabewert *gets* liefert bei fehlerfreier Ausführung die Stringvariable *s* zurück. Im Fehlerfall oder wenn beim Lesen das Dateiende überschritten wird, ist der Rückgabewert NULL.

Siehe auch *cgets*, *ferror*, *fgets*, *fopen*, *fputs*, *fread*, *freopen*, *getc*, *puts*, *scanf*

gettext — conio.h

Funktion Kopiert Text vom Textbildschirm in den Speicher.

Syntax

```
int gettext(int left, int top, int right, int bottom, void *destin);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *gettext* speichert den Inhalt eines rechteckigen Textbildschirmausschnitts in dem Speicherbereich, auf den *destin* zeigt. Die linke obere Ecke des Ausschnitts ist durch (*left*, *top*), die rechte untere Ecke durch (*bottom*, *right*) festgelegt.

Die Funktion arbeitet unabhängig von einem eventuell definierten Textfenster mit absoluten Bildschirmkoordinaten, wobei die obere linke Koordinatenecke mit (1,1) angesprochen wird.

Der Inhalt des Rechtecks wird von *gettext* sequentiell von links nach rechts und von oben nach unten in den Speicher eingelesen.

Jede Bildschirmposition benötigt zwei Bytes Speicherplatz: Das erste Byte ist das Zeichen selbst, das zweite das Video-Attribut an der jeweiligen Position. Der benötigte Speicherplatz für ein Rechteck mit *w* Spalten und *h* Zeilen ergibt sich aus:

Bytes = (*h* Zeilen) * (*w* Spalten) * 2

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert

Bei fehlerfreier Ausführung liefert *gettext* den Wert 1, im Fehlerfall (wenn Sie z.B. Koordinaten außerhalb des aktuellen Bildschirmmodus angegeben haben) 0 zurück.

Siehe auch

movetext, *puttext*

gettextinfo — conio.h

Funktion

Ermittelt Informationen über den aktuellen Textmodus.

Syntax

```
void gettextinfo(struct text_info *r);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

gettextinfo füllt die Elemente der mit *r* übergebenen Struktur des Typs *text_info* mit den aktuellen Parametern für Textausgaben.

Der Strukturtyp *text_info* ist in conio.h folgendermaßen definiert:

```
struct text_info {
   unsigned char winleft;      /* linke Fensterkoordinate */
   unsigned char wintop;       /* obere Fensterkoordinate */
   unsigned char winright;     /* rechte Fensterkoordinate */
   unsigned char winbottom;    /* untere Fensterkoordinate */
   unsigned char attribute;    /* Textattribut */
   unsigned char normattr;     /* normales Attribut */
   unsigned char currmode;     /* BW40, BW80, C40, C80 oder C4350 */
   unsigned char screenheight; /* Höhe des Textfensters */
   unsigned char screenwidth;  /* Breite des Textfensters */
   unsigned char curx,         /* Cursorposition im */
   unsigned char cury;         /* aktuellen Fenster */
};
```

➥ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner. Die Ergebnisse werden in der Struktur, auf die *r* zeigt, zurückgeliefert.

Siehe auch *textattr*, *textbackground*, *textcolor*, *textmode*, *wherex*, *wherey*, *window*

gettime, settime — dos.h

Funktion Lesen und Setzen der Systemzeit.

Syntax

```
void gettime(struct time *timep);
void settime(struct time *timep);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
gettime	■		■	■			■
settime	■		■				■

Beschreibung *gettime* liefert in der über *timep* angegebenen Struktur des Typs *time* die aktuelle Uhrzeit des Systems zurück.

settime setzt die Uhrzeit des Systems auf die Werte, die in der über *timep* angegebenen Struktur des Typs *time* enthalten sind.

Der Strukturtyp *time* ist in dos.h folgendermaßen definiert:

```
struct time {
   unsigned char ti_min;     /* Minuten */
   unsigned char ti_hour;    /* Stunden */
   unsigned char ti_hund;    /* 1/100 Sekunden */
   unsigned char ti_sec;     /* Sekunden */
};
```

Rückgabewert Keiner.

Siehe auch *_dos_gettime*, *_dos_settime*, *getdate*, *setdate*, *stime*, *time*

getvect, setvect — dos.h

Funktion Lesen und Setzen eines Interrupt-Vektors.

Syntax

```
void interrupt(*getvect(int interruptno))();                /* C-Version */
void interrupt(*getvect(int interruptno))( ... );           // C++ version
void setvect(int interruptno, void interrupt (*isr)());     /* C-Version */
void setvect(int interruptno, void interrupt (*isr)( ... )); // C++ Version
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Die Prozessoren der 8086-Familie reservieren eine Reihe von Interrupt-Vektoren, die von 0 bis 255 durchnumeriert sind. Der vier Byte lange Wert jedes Vektors ist die Adresse einer Interrupt-Funktion.

getvect liefert den Wert des Vektors *interruptno* als **far**-Zeiger auf eine Interrupt- Funktion zurück. Der Wert von *interruptno* muß zwischen 0 und 255 liegen.

setvect setzt den Vektor mit der Nummer *interruptno* so, daß er auf die als *isr* (interrupt service routine) angegebene C-Funktion zeigt. Diese C-Funktion muß als **interrupt**-Routine deklariert worden sein.

In C++ können nur statische Elementfunktionen und globale Funktionen als Interrupt-Funktionen deklariert werden.

Solange die in dos.h deklarierten Prototypen verwendet werden, ist *setvect* ohne weitere Modifikationen durch Übergabe der Adresse einer Interrupt-Funktion in jedem Speichermodell anwendbar.

Rückgabewert

getvect liefert den aktuellen, vier Byte langen Wert zurück, der im Interrupt-Vektor *interruptno* gespeichert ist.

setvect liefert keinen Wert zurück.

Siehe auch

disable, *_dos_getvect*, *_dos_setvect*, *enable*, *geniterrupt*

getverify — dos.h

Funktion

Liefert den Status des Verify-Flag des Betriebssystems.

Syntax

```
int getverify(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung

getverify liefert den aktuellen Status des Verify-Flag zurück.

Über das Verify-Flag wird festgelegt, ob Schreiboperationen auf der Diskette/Festplatte automatisch überprüft werden oder nicht. Wenn dieses Flag gesetzt ist, werden alle Schreiboperationen auf ihre Korrektheit überprüft.

Rückgabewert *getverify* liefert den aktuellen Status des Verify-Flag: 0 (Überprüfung ein) oder 1 (Überprüfung aus).

getw — stdio.h

Funktion Liest einen Integerwert aus einem Stream.

Syntax

```
int getw(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *getw* liest den nächsten Integer aus dem durch *stream* bezeichneten Eingabe-Stream. Eine spezielle Anordnung des Dateiinhalts wird nicht vorausgesetzt.

getw sollte nicht benutzt werden, wenn der Stream im Textmodus geöffnet wurde.

Rückgabewert *getw* liest zwei Bytes aus der angegebenen Datei und liefert sie als **int** zurück. Bei Überschreitung des Dateiendes ist der Rückgabewert EOF. Da EOF aber auch einen gültigen Wert darstellen kann, sollte ein expliziter Test mit *feof* oder *ferror* folgen, um auf Dateiende oder Fehler zu prüfen.

Siehe auch *putw*

gmtime — time.h

Funktion Rechnet Datum und Uhrzeit in die Greenwich Mean Time (GMT) um.

Syntax

```
struct tm *gmtime(const time_t *timer);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *gmtime* erwartet die Adresse eines Wertes, der von *time* zurückgeliefert wird, und liefert einen Zeiger auf eine Struktur vom Typ *tm*, in der die einzelnen Komponenten der Uhrzeit gespeichert sind.

Die globale **long**-Variable *_timezone* sollte den Unterschied zwischen der lokalen Zeit und der GMT in Sekunden enthalten (für MEZ ist *_timezone = -1 * 60 * 60*). Die globale Variable *_daylight* darf nur dann auf einen Wert ungleich null gesetzt werden, wenn die Standardkonvertierung der USA für Sommer- und Winterzeit angewendet werden soll.

Der Strukturtyp *tm* ist in time.h folgendermaßen definiert:

```
struct tm {
   int tm_sec;    /* Sekunden */
   int tm_min;    /* Minuten */
   int tm_hour;   /* Stunde (0 bis 23) */
   int tm_mday;   /* Tag im Monat (1 bis 31) */
   int tm_mon;    /* Monat (0 bis 11) */
   int tm_year;   /* Jahr (Kalenderjahr minus 1900) */
   int tm_wday;   /* Wochentag (0 bis 6, Sonntag = 0) */
   int tm_yday;   /* Tag im Jahr (0 bis 365) */
   int tm_isdst;  /* Ungleich null bei US-Sommerzeitkonvertierung */
};
```

Die Tageszeit wird in 24 Stunden gemessen, der Tag im Monat von 1 bis 31, der Monat von 0 bis 11 und der Tag im Jahr von 0 bis 365. Der Wochentag beginnt mit Sonntag = 0 und von der Jahresangabe muß 1900 subtrahiert werden. Das Flag *tm_isdst* gibt an, ob die Sommerzeitkonvertierung durchgeführt wurde.

Rückgabewert *gmtime* liefert einen Zeiger auf eine statische Struktur mit der ermittelten Information zurück, die durch jeden Aufruf dieser Funktion überschrieben wird.

G

Siehe auch *asctime*, *ctime*, *ftime*, *localtime*, *stime*, *time*, *tzset*

gotoxy — conio.h

Funktion Positioniert den Cursor im Textfenster.

Syntax `void gotoxy(int x, int y);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *gotoxy* positioniert den Cursor auf die angegebene Position im aktuellen Textfenster. Wenn die Koordinaten in irgendeiner Weise ungültig sind, wird der Aufruf von *gotoxy* vollständig ignoriert. Dies ist z.B. dann der Fall, wenn *gotoxy*(40,30) angegeben wird, die Koordinaten der rechten unteren Fensterecke aber bei (35,25) liegen.

Keines der beiden Argumente von *gotoxy* darf gleich 0 sein.

➥ Diese Funktion sollte in GUI-Anwendungen unter Win32s oder Win32 nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *wherex*, *wherey*, *window*

_heapadd malloc.h

Funktion Erweitert den Heap um einen Speicherblock.

Syntax

```
int _heapadd(void *block, size_t size);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung *_heapadd* erweitert den Heap um einen neuen Speicherblock, der über den Parameter *block* angegeben wird. Der Speicherblock darf nicht zuvor auf dem Heap reserviert worden sein. *_heapadd* wird normalerweise eingesetzt, um dem Heap große, statische Datenbereiche hinzuzufügen.

Rückgabewert *_heapadd* liefert bei fehlerfreier Ausführung 0, im Fehlerfall -1 zurück.

free, malloc

heapcheck alloc.h

Funktion Überprüft den Heap.

Syntax

```
int heapcheck(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *heapcheck* überprüft jeden Heap-Block auf seine Zeiger, Größe und weitere wichtige Attribute.

Rückgabewert Bei fehlerfreier Ausführung wird ein Wert größer als 0, im Fall eines Fehlers ein Wert kleiner als 0 zurückgeliefert. Im einzelnen können von der Funktion folgende Werte zurückgeliefert werden:

_HEAPCORRUPT	Der Heap ist ungültig
_HEAPEMPTY	Es gibt keinen Heap
_HEAPOK	Kein Fehler gefunden

heapcheckfree alloc.h

Funktion Überprüft die freien Blöcke des Heap auf einen konstanten Wert.

Syntax `int heapcheckfree(unsigned int fillvalue);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *heapcheckfree* überprüft die freien Blöcke des Heap auf einen konstanten Wert.

Rückgabewert Bei erfolgreicher Ausführung ist der Rückgabewert größer als 0, im Fall eines Fehlers kleiner als 0. Im einzelnen können von der Funktion folgende Werte zurückgeliefert werden:

_BADVALUE	Ein anderer Wert wurde gefunden
_HEAPCORRUPT	Der Heap ist ungültig
_HEAPEMPTY	Es gibt keinen Heap
_HEAPOK	Kein Fehler gefunden

heapchecknode — alloc.h

Funktion Überprüft einen einzelnen Knoten des Heap.

Syntax `int heapchecknode(void *node);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *heapchecknode* überprüft einzelne Knoten auf dem Heap. Wird *heapchecknode* nach der Freigabe eines Knotens mit einem Zeiger auf diesen Block aufgerufen, kann _BADNODE anstelle des erwarteten _FREEENTRY zurückgeliefert werden. Der Grund liegt darin, daß benachbarte freie Blöcke auf dem Heap zusammengefaßt werden und der betreffende Block deshalb nicht mehr existiert.

Rückgabewert Folgende Werte können zurückgeliefert werden:

_BADNODE	Der Knoten wurde nicht gefunden
_FREEENTRY	Der Knoten ist ein freier Block
_HEAPCORRUPT	Der Heap ist ungültig
_HEAPEMPTY	Es gibt keinen Heap
_USEDENTRY	Der Knoten ist ein reservierter Block

_heapchk — malloc.h

Funktion Überprüft den Heap.

Syntax `int _heapchk(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung Die Funktion *_heapchk* durchläuft den Heap, untersucht jeden Block und überprüft seine Zeiger, seine Größe und weitere wichtige Attribute.

Rückgabewert Einer der folgenden Werte wird zurückgeliefert:

_HEAPBADNODE	Ein fehlerhafter Block wurde gefunden
_HEAPEMPTY	Es gibt keinen Heap
_HEAPOK	Der Heap scheint fehlerfrei zu sein

heapfillfree — alloc.h

Funktion Füllt die freien Blöcke des Heap mit einem konstanten Wert.

Syntax `int heapfillfree(unsigned int fillvalue);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Rückgabewert Einer der folgenden Werte wird zurückgeliefert:

_HEAPCORRUPT	Der Heap ist ungültig
_HEAPEMPTY	Es gibt keinen Heap
_HEAPOK	Kein Fehler gefunden

_heapmin — malloc.h

Funktion Gibt unbenutzte Heap-Bereiche wieder frei.

Syntax `int _heapmin(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Funktion *_heapmin* gibt unbenutzte Heap-Bereiche wieder für das Betriebssystem frei. Dadurch können reservierte und danach wieder freigegebene Speicherblöcke von anderen Prozessen genutzt werden. Es ist kein Fehler, wenn wegen der Fragmentierung des Heap nicht immer alle unbenutzten Speicherblöcke an das Betriebssystem zurückgegeben werden können.

Rückgabewert *_heapmin* liefert bei erfolgreicher Ausführung 0, im Fehlerfall -1 zurück

Siehe auch *free*, *malloc*

_heapset — malloc.h

Funktion Füllt die freien Blöcke des Heap mit einem konstanten Wert.

Syntax

```
int _heapset(unsigned int fillvalue);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung *_heapset* überprüft den Zustand des Heap und verwendet dabei dieselben Methoden wie die Funktion *_heapchk*. Danach werden alle freien Blöcke des Heap mit dem Inhalt des niederwertigen Byte von *fillvalue* gefüllt. Die Funktion kann bei der Lösung von Problemen hilfreich sein, deren Ursache mit dem Heap zu tun hat. Sie garantiert jedoch *nicht*, daß später reservierte Speicherblöcke mit dem angegebenen Wert gefüllt sind.

Rückgabewert Einer der folgenden Werte wird zurückgeliefert:

_HEAPOK	Kein Fehler gefunden
_HEAPEMPTY	Es gibt keinen Heap
_HEAPBADNODE	Ein fehlerhafter Block wurde gefunden

Rückgabewert *_heapchk*, *_rtl_heapwalk*

G

heapwalk — alloc.h

Funktion Durchläuft den Heap Knoten für Knoten.

Syntax

```
int heapwalk(struct heapinfo *hi);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *heapwalk* durchläuft den Heap Knoten für Knoten. Die Funktion geht davon aus, daß der Heap gültig ist (überprüfen Sie dies vorher mit *heapcheck*). Mit dem letzten Block wird _HEAPOK zurückgeliefert. Bei einem erneuten Aufruf von *heapwalk* ist das Ergebnis _HEAPEND.

Die Funktion *heapwalk* übernimmt einen Zeiger auf eine Struktur vom Typ *heapinfo* (deklariert in alloc.h). Setzen Sie das Feld *hi.ptr* beim ersten Aufruf der Funktion auf NULL. Nach der Rückkehr enthält *hi.ptr* die Adresse des ersten Blocks, während *hi.size* die Größe des Blocks in Bytes angibt. Wird der Block zur Zeit benutzt, ist das Flag *hi.in_use* gesetzt.

Rückgabewert Einer der folgenden Werte wird zurückgeliefert:

_HEAPEMPTY	Es gibt keinen Heap
_HEAPEND	Das Ende des Heap wurde erreicht

_HEAPOK	Der *heapinfo*-Block enthält gültige Daten

_heapwalk — malloc.h

Veraltete Funktion. Siehe *_rtl_heap_walk*.

highvideo — conio.h

Funktion Setzt hohe Intensität für nachfolgende Textausgaben.

Syntax

```
void highvideo(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *highvideo* setzt das Bit für »hohe Intensität« der aktuellen Vordergrundfarbe und schaltet damit »hohe Intensität« für nachfolgende Textausgaben ein.

Bereits ausgegebene Zeichen bleiben unverändert – es wird lediglich das Zeichenattribut für Funktionen wie z.B. *cprintf* festgelegt, die direkte Bildschirmausgaben durchführen, *nachdem* die Funktion *highvideo* aufgerufen wurde.

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *cprintf*, *cputs*, *gettextinfo*, *lowvideo*, *normvideo*, *textattr*, *textcolor*

hypot, hypotl — math.h

Funktion Berechnet die Hypotenuse eines rechtwinkligen Dreiecks.

Syntax

```
double hypot(double x, double y);
long double hypotl(long double x, long double y);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
hypot	■	■	■	■			■
hypotl	■		■	■			■

Beschreibung *hypot* berechnet den Wert z, wobei gilt:

$z^2 = x^2 + y^2$ sowie $z \geq 0$.

Dies entspricht der Länge der Hypotenuse eines rechtwinkligen Dreiecks, dessen Kathete und Ankathete die Längen *x* bzw. *y* haben.

hypotl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert

Bei fehlerfreier Ausführung wird *z* als **double** (*hypot*) bzw. **long double** (*hypotl*) zurückgeliefert. Im Falle eines Fehlers (zum Beispiel bei einem Überlauf) liefert *hypot* HUGE_VAL und *hypotl* _LHUGE_VAL zurück, und die globale Variable *errno* wird auf folgenden Wert gesetzt:

ERANGE Ergebnis außerhalb des gültigen Bereichs

Die Fehlerbehandlung für *hypot* und *hypotl* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

_InitEasyWin io.h

Funktion

Initialisiert Easy Windows.

Syntax

```
void _InitEasyWin(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
		■				

Beschreibung

_InitEasyWin ermöglicht es 16 Bit Windows-Programmen, mit normalen Ein-/Ausgabefunktionen für Textbildschirme zu arbeiten.

Rückgabewert

Keiner.

inp conio.h

Funktion

Liest ein Byte aus einer Schnittstelle.

Syntax

```
int inp(unsigned portid);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Das Makro *inp* liest ein einzelnes Byte von der durch *portid* angegebenen Eingabeschnittstelle. *inp* ist als Inline-Makro definiert – der Compiler verwendet nur dann echte Funktionsaufrufe, wenn conio.h nicht aufgenommen wird (oder ein entsprechendes `#undef` stattgefunden hat).

Rückgabewert

inp liefert den gelesenen Wert zurück.

Siehe auch

inpw, *outp*, *outpw*

inport dos.h

Funktion Liest ein Wort aus einer Schnittstelle.

Syntax `int inport(int portid);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *inport* liest das niederwertige Byte eines Worts aus der durch *portid* angegebenen Schnittstelle und das höherwertige Byte aus der durch *portid + 1* angegebenen Adresse. Diese Funktion arbeitet wie die 80x86-Anweisung *IN*.

Rückgabewert *inport* liefert den gelesenen Wert zurück.

Siehe auch *inportb, outport, outportb*

inportb dos.h

Funktion Liest ein Byte aus einer Schnittstelle.

Syntax `unsigned char inportb(int portid);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *inportb* ist ein Makro, das einzelne Bytes von der durch *portid* angegebenen Schnittstelle einliest.

inportb ist als Inline-Makro definiert – der Compiler verwendet nur dann echte Funktionsaufrufe, wenn dos.h nicht aufgenommen wird oder wenn ein entsprechendes `#undef` stattgefunden hat.

Rückgabewert *inportb* liefert den gelesenen Wert zurück.

Siehe auch *import, outport, outportb*

inpw conio.h

Funktion Liest ein Wort aus einer Hardware-Schnittstelle.

Syntax `unsigned inpw(unsigned portid);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung Das Makro *inpw* liest ein 16-Bit Wort von der durch *portid* angegebenen Schnittstelle. *inpw* liest das niederwertige Byte des Wortes von *portid* und das höherwertige von *portid +1*.

inpw ist als Inline-Makro definiert – der Compiler verwendet nur dann echte Funktionsaufrufe, wenn conio.h nicht aufgenommen wird oder ein entsprechendes `#undef` stattgefunden hat.

Rückgabewert *inpw* liefert den gelesenen Wert zurück.

Siehe auch *inp*, *outp*, *outpw*

insline — conio.h

Funktion Fügt eine Leerzeile an der Cursor-Position in das Textfenster ein.

Syntax `void insline(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *insline* verschiebt alle Zeilen zwischen der aktuellen Cursor-Position und dem Ende des Textfensters um eine Zeile nach unten; die unterste Zeile wird dabei aus dem Fenster geschoben. Die neue Zeile wird in der aktuellen Hintergrundfarbe dargestellt.

➥ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

insline arbeitet nur im Textmodus.

Rückgabewert Keiner.

Siehe auch *clreol*, *delline*, *window*

int86 — dos.h

Funktion Allgemeiner 8086 Software-Interrupt.

Syntax `int int86(int intno, union REGS *inregs, union REGS *outregs);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *int86* führt einen 8086 Software-Interrupt durch, d.h. einen Aufruf der Routine, auf welche der durch *intno* angegebene Interrupt-Vektor zeigt.

Vor der Ausführung des Interrupts werden die Prozessorregister mit den durch *inregs* definierten Werten geladen.

Nach der Ausführung werden die Registerinhalte in *outregs*, der Status des Carry-Flag im Feld *x.cflag* von *outregs* und das 8086-Flag-Register im Feld *x.flags* von *outregs* gespeichert. Ein gesetztes Carry-Flag zeigt im allgemeinen einen Fehler an.

inregs und *outregs* können auf dieselbe Struktur zeigen.

Rückgabewert *int86* liefert den Wert des Registers AX nach der Ausführung des Interrupt zurück. Wenn das Carry-Flag durch den Interrupt gesetzt wurde (*outregs->x.cflag != 0*), so wird der globalen Variablen *_doserrno* der entsprechende Fehlercode zugewiesen. Beachten Sie, daß bei *gelöschtem* Carry-Flag (*outregs -> x.cflag == 0*) ebenfalls ein Fehler aufgetreten sein kann. Überprüfen Sie daher zur Sicherheit immer *_doserrno*.

Siehe auch *bdos*, *bdosptr*, *geninterrupt*, *int86x*, *intdos*, *intdosx*, *intr*

int86x — dos.h

Funktion Allgemeine Schnittstelle für 8086 Software-Interrupts.

Syntax

```
int int86x(int intno, union REGS *inregs, union REGS *outregs,
           struct SREGS *segregs);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *int86x* führt einen Software-Interrupt durch, d.h. einen Aufruf der Routine, auf die der durch *intno* angegebene Interrupt-Vektor zeigt.Vor der Ausführung des Interrupt werden die Prozessorregister mit den durch *inregs* angegebenen Werten geladen.

Zusätzlich kopiert *int86x* die Werte von *segregs->ds* und *segregs->es* in die entsprechenden Register, bevor es den Interrupt ausführt. Auf diese Weise kann ein Programm, das **far**-Zeiger oder ein großes Speichermodell benutzt, angeben, welches Segment für den Software-Interrupt verwendet werden soll.

Nach der Rückkehr vom Software-Interrupt kopiert *int86x* die Registerwerte in *outregs*, den Status des Carry-Flag in das Feld *x.cflag* von *outregs*, und das 8086-Flag-Register in das Feld *x.flags* von *outregs*. Anschließend stellt *int86x* den alten Inhalt des DS-Registers wieder her und speichert die Werte der Segmentregister in den Feldern *segregs->es* bzw. *segregs->ds*. Ist das Carry-Flag gesetzt, zeigt das im allgemeinen einen Fehler an.

int86x ermöglicht den Aufruf eines 8086-Software-Interrupt, der mit einem anderen als dem voreingestellten Wert von DS arbeitet und/oder ein Argument in ES benötigt.

inregs und *outregs* können auf dieselbe Struktur zeigen.

Rückgabewert *int86x* liefert den Wert des Registers AX nach der Ausführung des Interrupt zurück. Wenn das Carry-Flag durch den Interrupt gesetzt wurde (*outregs->x.cflag != 0*), so wird der globalen Variablen *_doserrno* der entsprechende Fehlercode zugewiesen. Beachten Sie, daß bei *gelöschtem* Carry-Flag (*outregs -> x.cflag == 0*) ebenfalls ein Fehler aufgetreten sein kann. Überprüfen Sie daher zur Sicherheit immer *_doserrno*.

Siehe auch *bdos*, *bdosptr*, *geninterrupt*, *intdos*, *intdosx*, *int86*, *intr*, *segread*

intdos — dos.h

Funktion Allgemeine Schnittstelle für DOS-Interrupts.

Syntax

```
int intdos(union REGS *inregs, union REGS *outregs);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

I

Beschreibung *intdos* führt einen Aufruf des DOS-Interrupt 0x21 aus, wobei die Nummer der auszuführenden Funktion durch *inregs->h.ah* angegeben sein muß.

Nach der Rückkehr vom Interrupt 0x21 kopiert *intdos* die Registerwerte in *outregs*, den Status des Carry-Flag in das Feld *x.cflag* von *outregs* und den Inhalt des 8086-Flag-Register in das Feld *x.flags* von *outregs*. Ein gesetztes Carry-Flag zeigt einen Fehler an.

inregs und *outregs* können auf dieselbe Struktur zeigen.

Rückgabewert *intdos* liefert den von DOS gesetzten Inhalt des Registers AX zurück. Wenn das Carry-Flag durch den Aufruf gesetzt wurde (*outregs->x.cflag != 0*), dann wird der globalen Variablen *_doserrno* der entsprechende Fehlercode zugewiesen. Beachten Sie, daß bei *gelöschten* Carry-Flag (*outregs -> x.cflag == 0*) ebenfalls ein Fehler aufgetreten sein kann. Überprüfen Sie daher zur Sicherheit immer *_doserrno*.

Siehe auch *bdos*, *bdosptr*, *geninterrupt*, *int86*, *int86x*, *intdosx*, *intr*

intdosx — dos.h

Funktion Allgemeine Schnittstelle für DOS-Interrupts.

Syntax

```
int intdosx(union REGS *inregs, union REGS *outregs, struct SREGS *segregs);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

intdosx führt einen Aufruf des DOS-Interrupt 0x21 aus, wobei die Nummer der auszuführenden Funktion durch *inregs->h.ah* angegeben sein muß.

Zusätzlich kopiert *intdosx* die Werte von *segregs->ds* und *segregs->es* in die entsprechenden Register, bevor es den Interrupt ausführt. Auf diese Weise kann ein Programm, das **far**-Zeiger oder ein großes Speichermodell benutzt, angeben, welches Segment für den Software-Interrupt verwendet werden soll.

Nach der Rückkehr vom Interrupt 0x21 kopiert *intdosx* die Registerwerte in *outregs*, den Status des Carry-Flag in das Feld *x.cflag* von *outregs* und das 8086-Flag-Register in das Feld *x.flags* von *outregs*. Anschließend stellt *int86x* den alten Inhalt des DS-Registers wieder her und speichert die Werte der Segmentregister in den Feldern *segregs->es* bzw. *segregs->ds*. Ist das Carry-Flag gesetzt, zeigt das im allgemeinen einen Fehler an.

intdosx ermöglicht den Aufruf eines 8086-Software-Interrupt, der mit einem anderen als dem voreingestellten Wert von DS arbeitet und/oder ein Argument in ES benötigt.

inregs und *outregs* können auf dieselbe Struktur zeigen.

Rückgabewert

intdosx liefert den von DOS gesetzten Inhalt des Registers AX zurück. Wenn das Carry-Flag durch den Aufruf gesetzt wurde (*outregs->x.cflag != 0*), wird der globalen Variablen *_doserrno* der entsprechende Fehlercode zugewiesen. Beachten Sie, daß bei *gelöschtem Carry-Flag (outregs -> x.cflag = 0*) ebenfalls ein Fehler aufgetreten sein kann. Überprüfen Sie daher zur Sicherheit immer *_doserrno*.

Siehe auch

bdos, *bdosptr*, *geninterrupt*, *int86*, *int86x*, *intdos*, *intr*, *segread*

intr — dos.h

Funktion

Alternative Schnittstelle für 8086-Software-Interrupts.

Syntax

```
void intr(int intno, struct REGPACK *preg);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

Die Funktion *intr* ist eine alternative Schnittstelle für die Ausführung von Interrupts. Sie erzeugt den vom Parameter *intno* angegebenen 8086 Software-Interrupt.

Vor der Ausführung des Interrupts werden die Prozessorregister auf die Werte der *REGPACK*-Struktur gesetzt, auf die der Parameter *preg* zeigt. Nach der Ausführung erfolgt eine Kopie der Registerinhalte (und der Flags) in die Struktur **preg*.

Folgende Parameter werden der Funktion *intr* übergeben:

intno Nummer des auszuführenden Interrupts.

preg Adresse einer Struktur mit

(a) den Registerinhalten vor dem Aufruf
(b) den Registerinhalten nach dem Aufruf

Der Strukturtyp REGPACK ist in dos.h folgendermaßen definiert:

```
struct REGPACK {
   unsigned r_ax, r_bx, r_cx, r_dx;
   unsigned r_bp, r_si, r_di, r_ds, r_es, r_flags;
};
```

Rückgabewert Keiner. Die REGPACK-Struktur **preg* enthält die Registerwerte nach dem Aufruf des Interrupt.

Siehe auch *geninterrupt*, *int86*, *int86x*, *intdos*, *intdosx*

I

ioctl io.h

Funktion Steuerung von Peripheriegeräten.

Syntax `int ioctl(int handle, int func[, void *argdx, int argcx]);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■				

Beschreibung *ioctl* ist im Prinzip auf UNIX-Systemen verfügbar – sowohl die erwarteten Parameter als auch die ausgeführten Aktionen sind allerdings anders definiert. Selbst innerhalb von UNIX arbeitet diese Funktion nicht einheitlich – System III und die Version 7 implementieren sie auf verschiedene Weise. Die DOS-Funktion *ioctl* ist deshalb nicht nach UNIX portierbar (und nur eingeschränkt portierbar zwischen verschiedenen DOS-Versionen).

Unter der DOS-Version 3.0 sind zusätzlich die Werte 8 und 11 für *func* definiert.

ioctl verwendet die DOS-Funktion 0x44, wobei die tatsächlich ausgeführten Aktionen sehr stark von *func* abhängen.

Die folgenden Werte sind für *func* definiert:

0 Lesen von Informationen über das Gerät.
1 Setzen von Geräteinformationen durch den Parameter *argdx*.
2 Lesen von *argcx* Bytes vom Gerät in den Bereich, auf den *argdx* zeigt.
3 Schreiben von *argcx* Bytes zu dem Gerät; *argdx* zeigt auf die Start-adresse
im Speicher.
4 Wie Unterfunktion 2, allerdings wird der Wert von *handle* hier als Laufwerksnummer interpretiert (0 = Standard, 1 = A, 2 = B, etc.).
5 Wie Unterfunktion 3, allerdings mit *handle* als Laufwerksnummer (0 = Standard, 1 = A, 2 = B usw.).
6 Ermittlung des Eingabestatus.
7 Ermittlung des Ausgabestatus.
8 Prüfung, ob der Datenträger auswechselbar ist (nur DOS-Version 3.0).
11 Setzen der Anzahl der Wiederholversuche bei Kollisionen beim gemeinsamen Zugriff (nur DOS-Version 3.0).

ioctl wird normalerweise für Geräteeinheiten verwendet, kann aber auch zusammen mit normalen Diskettendateien eingesetzt werden. Im letzteren Fall sind allerdings nur die Unterfunktionen 0, 6 und 7 definiert – alle anderen Aufrufe liefern einen Fehler zurück und setzen *errno* auf den Wert EINVAL.

Genaue Informationen zur DOS-Funktion 0x44 (IOCTL), deren Parameter und Rückgabewerte finden Sie in Ihrem DOS-Referenzhandbuch.

Die Argumente *argdx* und *argcx* sind optional.

ioctl stellt eine direkte Schnittstelle zu speziellen Funktionen von DOS-Gerätetreibern dar. Das exakte Verhalten dieser Funktion hängt daher stark von der verwendeten Hardware und den angesprochenen Geräten ab. Nicht alle Hardware-Hersteller halten sich an die Schnittstelle, wie sie hier beschrieben wurde. Für die korrekte Benutzung von *ioctl* benötigen Sie die BIOS-Dokumentation des Herstellers.

Rückgabewert

Für die Werte 0 und 1 von *func* liefert *ioctl* den Inhalt des Prozessorregisters DX zurück – die gelesene/gesetzte Geräteinformation. Für Werte von *func* im Bereich von 2 bis 5 wird die Anzahl der tatsächlich gelesenen/geschriebenen Bytes zurückgeliefert. Für die Werte 6 und 7 wird die Statusinformation zurückgeliefert.

Im Falle eines Fehlers liefert *ioctl* den Wert -1 und setzt *errno* auf einen der folgenden Werte:

EBADF	Ungültiges Handle
EINVAL	Ungültiges Argument
EINVDAT	Ungültige Daten

isalnum ctype.h

Funktion Testet, ob ein Zeichen ein alphanumerisches Zeichen ist.

Syntax `int isalnum(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Das Makro *isalnum* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. *isalnum* ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* ein Buchstabe (*A* bis *z* oder *a* bis *z*) oder eine Ziffer (*0* bis *9*).

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isalnum* liefert einen Wert ungleich null zurück, wenn *c* ein Buchstabe oder eine Ziffer ist.

isalpha ctype.h

Funktion Testet, ob ein Zeichen ein Buchstabe ist.

Syntax `int isalpha(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *isalpha* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* ein Buchstabe (*A* bis *Z* oder *a* bis *z*).

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isalpha* liefert einen Wert ungleich null zurück, wenn *c* ein Buchstabe ist.

isascii ctype.h

Funktion Makro zur Klassifizierung von Zeichen.

Syntax `int isascii(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *isascii* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro liefert einen Wert ungleich null für wahr und den Wert 0 für falsch zurück.

isascii ist für alle Integerwerte definiert.

Rückgabewert *isascii* liefert einen Wert ungleich null zurück, wenn das niederwertige Byte von *c* im Bereich von 0 bis 127 (0x00-0x7F) liegt.

isatty — io.h

Funktion Prüft den Gerätetyp.

Syntax

```
int isatty(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *isatty* prüft, ob das über *handle* angegebene Peripheriegerät zu einem der folgende Gerätetypen gehört:

- Terminal
- Tastatur
- Drucker
- serielle Schnittstelle

Rückgabewert Wenn das Gerät eines der oben aufgeführten Geräte ist, so liefert *isatty* einen Wert ungleich null. Andernfalls wird 0 zurückgeliefert.

iscntrl — ctype.h

Funktion Testet, ob ein Zeichen ein Steuerzeichen ist.

Syntax

```
int iscntrl(int c);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *iscntrl* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* ein Löschzeichen oder ein Steuerzeichen (0x7F oder 0x00 bis 0x1f).

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *iscntrl* liefert einen Wert ungleich null zurück, wenn *c* ein Steuerzeichen oder ein Löschzeichen ist.

isdigit ctype.h

Funktion Testet, ob ein Zeichen eine Dezimalziffer ist.

Syntax `int isdigit(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *isdigit* prüft das übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* eine Ziffer (*0* bis *9*).

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isdigit* liefert einen Wert ungleich null zurück, wenn *c* eine Ziffer ist.

I

isgraph ctype.h

Funktion Testet, ob ein Zeichen ein druckbares Zeichen ist.

Syntax `int isgraph(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *isgraph* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* ein druckbares Zeichen mit Ausnahme des Leerzeichens (' ').

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isgraph* liefert einen Wert ungleich null zurück, wenn *c* ein druckbares Zeichen ist.

islower ctype.h

Funktion Testet, ob ein Zeichen ein Kleinbuchstabe ist.

Syntax `int islower(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *islower* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* ein Kleinbuchstabe (*a* bis *z*).

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *islower* liefert einen Wert ungleich null zurück, wenn *c* ein Kleinbuchstabe ist.

isprint ctype.h

Funktion Testet, ob ein Zeichen ein druckbares Zeichen ist.

Syntax `int isprint(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *isprint* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* ein druckbares Zeichen einschließlich des Leerzeichens (' ').

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isprint* liefert einen Wert ungleich null zurück, wenn *c* ein druckbares Zeichen ist.

ispunct ctype.h

Funktion Testet, ob ein Zeichen ein Interpunktionszeichen ist.

Syntax `int ispunct(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *ispunct* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE ab-

hängig. In der Standardeinstellung C ist *c* jedes druckbare Zeichen, das weder ein alphanumerisches Zeichen noch ein Leerzeichen (' ') ist.

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *ispunct* liefert einen Wert ungleich null zurück, wenn *c* ein Interpunktionszeichen ist.

isspace — ctype.h

Funktion Testet, ob ein Zeichen ein Leerzeichen ist.

Syntax `int isspace(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *isspace* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig.

Sie können das Makro durch eine entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isspace* liefert einen Wert ungleich null zurück, wenn *c* ein Leerzeichen, Tabulator, Wagenrücklauf, Zeilenvorschub, vertikaler Tabulator oder ein Seitenvorschub (ASCII-Codes 0x09 bis 0x0D, 0x20) oder jedes andere in der Locale-Kategorie definierte Leerzeichen ist.

isupper — ctype.h

Funktion Testet, ob ein Zeichen ein Großbuchstabe ist.

Syntax `int isupper(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *isupper* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig. In der Standardeinstellung C ist *c* ein Großbuchstabe (*A* bis *Z*).

Sie können das Makro durch ein entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isupper* liefert einen Wert ungleich null zurück, wenn *c* ein Großbuchstabe ist.

isxdigit — ctype.h

Funktion Testet, ob ein Zeichen eine hexadezimale Ziffer ist.

Syntax

```
int isxdigit(int c);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *isxdigit* prüft das als Integer übergebene ASCII-Zeichen anhand einer Tabelle. Das Makro ist von der aktuellen Locale-Kategorie LC_CTYPE abhängig.

Sie können das Makro durch eine entsprechendes `#undef` als Funktion verfügbar machen.

Rückgabewert *isxdigit* liefert einen Wert ungleich null zurück, wenn *c* eine hexadezimale Ziffer (0 bis 9, A bis F, a bis f) oder jede andere durch die Locale-Kategorie definierte hexadezimale Ziffer ist.

itoa — stdlib.h

Funktion Konvertiert einen Integerwert in einen String.

Syntax

```
char *itoa(int value, char *string, int radix);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *itoa* konvertiert den übergebenen Integerwert *value* in einen nullterminierten String und speichert das Ergebnis in der durch *string* bezeichneten Variablen.

Der Parameter *radix* legt die Basis fest, die bei der Konvertierung benutzt werden soll. Mögliche Werte liegen im Bereich von 2 bis 36. Wenn *radix* den Wert 10 hat und *value* kleiner als null ist, stellt *itoa* dem Ergebnis ein Minuszeichen voran.

➥ Der durch den Parameter *string* angegebene String muß lang genug sein, um das Ergebnis der Konvertierung inklusive eines abschließenden Nullzeichens (\0) aufnehmen zu können. *itoa* erzeugt maximal 17 Bytes.

Rückgabewert *itoa* liefert deren Zeiger *string* zurück.

Siehe auch *ltoa*, *utoa*

kbhit conio.h

Funktion Prüft auf verfügbare Tastenanschläge.

Syntax

```
int kbhit(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *kbhit* prüft, ob ein Zeichen über die Tastatur eingegeben wurde. Verfügbare Zeichen können mit *getch* oder *getche* gelesen werden.

➡ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Wenn eine noch nicht gelesene Tastatureingabe vorliegt, liefert *kbhit* einen Wert ungleich null. Liegt kein Zeichen an, dann liefert *kbhit* den Wert 0.

Siehe auch *getch*, *getche*

labs math.h

Funktion Liefert den absoluten Wert eines long-Wertes.

Syntax

```
long labs(long int x);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *labs* berechnet den absoluten Wert des **long**-Arguments *x*.

Rückgabewert *labs* liefert den absoluten Wert des **long**-Arguments *x* zurück.

Siehe auch *abs*, *caps*, *fabs*

K–L

ldexp, ldexpl math.h

Funktion Berechnet $x * 2^{\text{exp}}$.

Syntax

```
double ldexp(double x, int exp);
long double ldexpl(long double x, int exp);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
ldexp	■	■	■	■	■	■	■
ldexpl	■		■	■			■

Beschreibung

ldexp liefert das Ergebnis der Berechnung $x * 2^{exp}$ als **double**-Wert zurück.

ldexpl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert

Bei fehlerfreier Ausführung liefern *ldexp* und *ldexpl* das Ergebnis der Berechnung von $x * 2^{exp}$ zurück.

Die Fehlerbehandlung kann für diese Funktionen über die Routine _matherr bzw. *_matherrl* modifiziert werden.

Siehe auch

exp, frexp, modf

ldiv — stdlib.h

Funktion

Dividiert zwei **long**-Werte und liefert den Quotienten und den Divisionsrest zurück.

Syntax

```
ldiv_t ldiv(long int numer, long int denom)
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung

ldiv dividiert die als *numer* (Dividend) und *denom* (Divisor) übergebenen **long**-Werte und liefert eine Struktur des Typs *ldiv_t* zurück. Dieser Datentyp ist in stdlib.h folgendermaßen definiert:

```
typedef struct {
   long int quot;     /* Quotient */
   long int rem;      /* Divisionsrest */
} ldiv_t;
```

Rückgabewert

Es wird eine Struktur des zuvor beschriebenen Typs zurückgeliefert.

Siehe auch

div

lfind — stdlib.h

Funktion

Führt eine lineare Suche nach einem Array-Element durch.

Syntax

```
void *lfind(const void *key, const void *base, size_t *num, size_t width,
            int (_USERENTRY *fcmp)(const void *, const void *));
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

lfind durchsucht ein sequentiell angeordnetes Array nach dem Wert des Schlüsselfeldes *key*. Die Suche erfolgt linear. *lfind* verwendet eine vom Programmierer erstellte Vergleichsroutine *fcmp*, die mit der _USERENTRY Aufrufkonvention verwendet werden muß.

Der Parameter *base* muß auf das erste Element des Array zeigen, *num* auf einen Integer, der die Anzahl der Array-Elemente enthält. *width* gibt die Größe eines einzelnen Elements in Bytes an.

Rückgabewert

lfind liefert einen Zeiger auf das erste Element des Array zurück, das mit dem Suchschlüssel *key* identisch ist. Falls der gesuchte Eintrag nicht existiert, liefert *lfind* den Wert NULL. Die Vergleichsroutine muß für **elem1* == **elem2* (das sind ihre beiden Parameter) den Wert 0 liefern, andernfalls einen Wert ungleich null.

Siehe auch

bsearch, *lsearch*, *qsort*

localeconv — locale.h

Funktion

Ermittelt die Locale-Einstellung für numerische Formate.

Syntax

```
struct lconv *localeconv(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung

localeconv liefert Informationen über länderspezifische, numerische Formate (z.B. für die Währung). Diese Informationen sind in der Struktur des Typs *lconv* gespeichert und können nur über die Funktion *setlocale* geändert werden. Nachfolgende Aufrufe von *localeconv* aktualisieren die Struktur *lconv*.

Der Strukturtyp *lconv* ist in locale.h folgendermaßen definiert:

Tabelle 3.1: Locale-Einstellungen für Währungs- und numerische Formate

Feld	Anwendung
char **decimal_point*;	Normaler Dezimalpunkt (ohne Währungsformate). Darf niemals ein leerer String sein.
char **thousands_sep*;	Zeichen, um Ziffern links des Dezimalpunkts zu gruppieren. Wird nicht bei Währungsangaben verwendet.
char **grouping*;	Größe jeder Zifferngruppe. Wird nicht bei Währungsangaben verwendet. Siehe weiter untenstehende Wertetabelle.
char **int_curr_symbol*;	Internationales Währungssymbol der aktuellen Locale-Einstellung. Das Symbolformat ist in der Norm ISO 4217 »Codes for Representation of Currency and Funds« festgelegt.
char **currency_symbol*;	Lokales Währungssymbol der aktuellen Locale-Einstellung;
char **mon_decimal_point*;	Dezimalpunkt für Währungsangaben.
char **mon_thousands_sep*;	Zeichen, um bei Währungangaben Ziffern links des Dezimalpunkts zu gruppieren.
char **mon_grouping*;	Größe jeder Zifferngruppe bei Währungsangaben. Wird nicht bei Währungsangaben verwendet. Siehe weiter unten stehende Wertetabelle.
char **positive_sign*;	String, der nichtnegative Währungsangaben kennzeichnet.
char **negative_sign*;	String, der positive Währungsangaben kennzeichnet.
char *int_frac_digits*;	Anzahl von Ziffern hinter dem Dezimalpunkt, die bei einer international formatierten Währungsangabe dargestellt werden.
char *frac_digits*;	Anzahl von Ziffern hinter dem Dezimalpunkt, die bei einer formatierten Währungsangabe dargestellt werden.
char *p_cs_precedes*;	Ist auf 1 gesetzt, wenn *currency_symbol* einer nichtnegativen, formatierten Währungsangabe vorangestellt wird. Wird *currency_symbol* nachgestellt, ist das Feld auf 0 gesetzt.
char p_sep_by_space;	Ist auf 1 gesetzt, wenn *currency_symbol* von einer nichtnegativen, formatierten Währungsangabe durch ein Leerzeichen getrennt wird. Andernfalls wird das Feld auf 0 gesetzt.
char *n_cs_precedes*;	Ist auf 1 gesetzt, wenn *currency_symbol* einer negativen, formatierten Währungsangabe vorangestellt wird. Wird *currency_symbol* nachgestellt, so ist das Feld auf 0 gesetzt.
char *p_sep_by_space*;	Ist auf 1 gesetzt, wenn *currency_symbol* von einer negativen, formatierten Währungsangabe durch ein Leerzeichen getrennt wird. Andernfalls wird das Feld auf 0 gesetzt.
char *p_sign_posn*;	Legt fest, wo das Zeichen *positive_sign* für eine nichtnegative, formatierte Währungsangabe dargestellt werden soll.
char *p_sign_posn*;	Legt fest, wo das Zeichen *negative_sign* für eine negative, formatierte Währungsangabe dargestellt werden soll.

Alle obigen Strings mit Ausnahme von *decimal_point* werden in der aktuellen Locale-Einstellung nicht unterstützt. Die **char**-Felder enthalten nichtnegative Werte. Ist ein **char**-Feld auf *CHAR_MAX* gesetzt, so wird es in der aktuellen Locale-Einstellung nicht unterstützt.

Die Felder *grouping* und *mon_grouping* werden folgendermaßen gesetzt und interpretiert:

Wert	Bedeutung
CHAR_MAX	Es wird keine weitere Gruppierung durchgeführt.
0	Das vorherige Element soll für den Rest der Ziffern wiederholt verwendet werden.
jeder andere Integerwert	Legt die Größe der aktuellen Ziffergruppe fest. Um die Größe der folgenden Ziffergruppe zu bestimmen, wird das nächste Element eingelesen.

Die Felder *p_sign_posn* und *n_sign_posn* werden folgendermaßen gesetzt und interpretiert:

Wert	Bedeutung
0	Die Währungsangabe wird einschließlich des Währungssymbols *currency_symbol* eingeklammert.
1	Der Vorzeichenstring wird vor die Währungsangabe und *currency_symbol* gesetzt.
2	Der Vorzeichenstring wird hinter die Währungsangabe und *currency_symbol* gesetzt.
3	Der Vorzeichenstring wird direkt vor dir Währungsangabe und *currency_symbol* gesetzt.
4	Der Vorzeichenstring wird direkt hinter die Währungsangabe und *currency_symbol* gesetzt.

Rückgabewert *localeconv* liefert einen Zeiger auf die gefüllte Struktur des Typs *lconv* zurück. Die Strukturwerte werden bei jedem Aufruf von *setlocale*, der die Locale-Kategorien LC_MONETARY oder LC_NUMERIC ändert, aktualisiert.

Siehe auch *setlocale*

localtime — time.h

Funktion Konvertiert Datum und Uhrzeit in eine Struktur des Typs *tm*.

Syntax

```
struct tm *localtime(const time_t *timer);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *localtime* übernimmt die Adresse eines Wertes, der durch *time* geliefert wird, und gibt einen Zeiger auf die Struktur des Typs *tm* zurück, die die umgerechneten Zeitangaben enthält. *localtime* berücksichtigt hierbei Zeitzonen und die Sommerzeit.

Die globale **long**-Variable *_timezone* muß auf den Unterschied zwischen der lokalen Zeit und der *Greenwich Mean Time* (GMT) in Sekunden gesetzt werden (für MEZ ist *_timezone* -1*60*60). Die globale Variable *_daylight* darf nur dann auf einen Wert ungleich null gesetzt werden, wenn die Standardwerte der USA für Sommer- und Winterzeit angewendet werden können. Diese Werte werden von der Funktion *tzset* und nicht von einem Benutzerprogramm direkt gesetzt.

Der Strukturtyp *tm* ist in time.h folgendermaßen definiert:

```
struct tm {
   int tm_sec;     /* Sekunden */
   int tm_min;     /* Minuten */
   int tm_hour;    /* Stunden (0..23) */
   int tm_mday;    /* Tag im Monat (1..31) */
   int tm_mon;     /* Monat (0..11) */
   int tm_year;    /* Jahr (-1900) */
   int tm_wday;    /* Wochentag (0..6) */
   int tm_yday;    /* Tag im Jahr (0..365) */
   int tm_isdst;   /* Sommerzeit */
};
```

Die Tageszeit wird in 24 Stunden gemessen, der Tag im Monat von 1 bis 31, der Monat von 0 bis 11, und der Tag im Jahr von 0 bis 365. Der Wochentag beginnt mit Sonntag = 0 und von der Jahresangabe muß 1900 subtrahiert werden. Das Flag *tm_isdst* gibt an, ob die Sommerzeitkonvertierung durchgeführt wurde.

Rückgabewert *localtime* liefert einen Zeiger auf eine statische Struktur, die die einzelnen Zeitangaben enthält. Die Struktur wird durch jeden Aufruf dieser Funktionen erneut gesetzt.

Siehe auch *asctime*, *ctime*, *ftime*, *gmtime*, *stime*, *time*, *tzset*

lock — io.h

Funktion Sperrt Teile einer Datei im gemeinsamen Zugriff.

Syntax

```
int lock(int handle, long offset, long length);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *lock* bietet eine Schnittstelle zu den vom Betriebssystem bereitgestellten Verfahren für den gemeinsamen Dateizugriff (file-sharing).

lock kann beliebige Bereiche einer Datei sperren, die sich jedoch nicht überschneiden dürfen. Jedes Programm, das auf einen gesperrten Bereich

zugreifen will, wiederholt diese Operation dreimal und bricht dann mit einem Fehler ab.

Rückgabewert Die Funktion liefert bei fehlerfreier Ausführung den Wert 0. Andernfalls wird der Wert -1 zurückgeliefert und die globale Variable *errno* auf folgenden Wert gesetzt:

EACCES Zugriff nicht gestattet

Siehe auch *locking*, *open*, *sopen*, *unlock*

locking — io.h, sys\locking.h

Funktion Sperrt oder entsperrt Teile einer Datei im gemeinsamen Zugriff.

Syntax

```
int locking(int handle, int cmd, long length);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *locking* bietet eine Schnittstelle zu den vom Betriebssystem bereitgestellten Verfahren für den gemeinsamen Dateizugriff (file-sharing). Die Funktion sperrt *length* Bytes der durch *handle* bezeichneten offenen Datei, ab der durch *offset* vorgegebenen Position gegen Zugriffe anderer Programme.

Mit *locking* lassen sich beliebig viele Bereiche einer Datei sperren, die sich jedoch nicht überlappen dürfen. Jedes Programm, das auf einen gesperrten Bereich zugreifen will, wiederholt diese Operation dreimal und bricht dann mit einem Fehler ab.

cmd gibt die durchzuführende Operation an. Folgende Konstanten sind dafür in sys\locking.h definiert:

LK_LOCK	Sperren des Bereichs. Ist das Sperren nicht möglich, wird der Sperrversuch 10 Sekunden lang im 1-Sekunden-Takt wiederholt. Anschließend Abbruch.
LK_RLCK	Wie LK_LOCK.
LK_NBLCK	Sperren des Bereichs. Sofortiger Abbruch, wenn Sperrung nicht möglich ist.
LK_NBRLCK	Wie LK_NBLCK.
LK_UNLCK	Entsperren des Bereichs (der Bereich muß vorher gesperrt worden sein).

K–L

Rückgabewert

Die Funktion liefert bei fehlerfreier Ausführung den Wert 0. Andernfalls wird der Wert -1 zurückgeliefert und die globale Variable *errno* auf einen der folgenden Werte gesetzt:

EACCES	Datei bereits gesperrt bzw. entsperrt
EBADF	Ungültiges Datei-Handle
EDEADLOCK	Datei konnte nach 10 Versuchen nicht gesperrt werden (*cmd* ist LK_LOCK oder LK_RLCK)
EINVAL	Ungültiges *cmd* oder SHARE.EXE nicht geladen

Siehe auch

_fsopen, *lock*, *open*, *sopen*, *unlock*

log, logl — math.h

Funktion

Berechnet den natürlichen Logarithmus des Arguments.

Syntax

```
double log(double x);
long double logl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
log	■	■	■	■	■	■	■
logl	■		■	■			■

log liefert den natürlichen Logarithmus des Arguments *x* zurück.

logl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert

Bei fehlerfreier Ausführung liefern *log* und *logl* den berechneten Wert *ln(x)* zurück. Ist *x* ein reelles Argument und *x* < *0*, wird die globale Variable *errno* auf folgenden Wert gesetzt:

EDOM	Bereichsfehler

Ist *x* == *0*, dann liefern die Funktionen den Wert -HUGE_VAL (*log*) bzw. -_LHUGE_VAL (*logl*) zurück und setzen *errno* auf ERANGE. Die Fehlerbehandlung für *log* und *logl* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

Siehe auch

bcd, *complex*, *exp*, *log10*, *sqrt*

log10, log10l — math.h

Funktion Berechnet den Logarithmus des Arguments zur Basis 10.

Syntax

```
double log10(double x);
long double log10l(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
log10	■	■	■	■	■	■	■
log10l	■		■	■			■

Beschreibung *log10* berechnet den Logarithmus *n* des Arguments *x* zur Basis 10, erfüllt also die Gleichung $x = 10^n$.

log10l ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert *log10* und *log10l* liefern das Ergebnis der Berechnung von $log_{10}(x)$ zurück.

Ist *x* ein reelles Argument und $x < 0$, so setzen *log10* und *log10l* die globale Variable *errno* auf den Wert

EDOM Bereichsfehler

Ist $x == 0$, dann liefern die Funktionen den Wert -HUGE_VAL (*log10*) bzw. -_LHUGE_VAL (*log10l*) zurück. Die Fehlerbehandlung für *log10* und *log10l* kann über die Funktionen *_matherr* bzw. *_matherrl* modifiziert werden.

Siehe auch *bcd*, *complex*, *exp*, *log*

longjmp — setjmp.h

Funktion Führt ein nicht-lokales *goto* aus.

Syntax

```
void longjmp(jmp_buf jmpb, int retval);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *setjmp* speichert den aktuellen Task-Zustand in dem durch *jmpb* angegebenen Puffer. Ein Aufruf von *longjmp* zu einem späteren Zeitpunkt stellt diesen Zustand wieder her. Das Programm wird an dieser Stelle so fortgesetzt, als hätte *setjmp* den Wert *retval* zurückgeliefert.

Ein Task-Zustand beinhaltet:

Win16	Win32
alle Segmentregister (CS, DS, ES, SS)	Segmentregister werden nicht gesichert
Registervariablen (DI, SI)	Registervariablen (EBX, EDI, ESI)
Stack-Zeiger (SP)	Stack-Zeiger (ESP)
Frame-Zeiger (BP)	Frame-Zeiger (EBP)
Flags	Flags werden nicht gesichert

Ein Task-Zustand ist so vollständig, daß mit Hilfe von *setjmp* und *longjmp* Koroutinen implementiert werden können.

setjmp muß vor *longjmp* aufgerufen werden. Die Routine, die *setjmp* aufgerufen und *jmpb* einrichtet hat, muß beim Aufruf von *longjmp* noch aktiv sein und darf daher nicht zwischenzeitlich beendet worden sein. Andernfalls sind die Ergebnisse nicht vorhersehbar.

longjmp kann den Wert 0 nicht übergeben. Wird *retval* auf 0 gesetzt, so ersetzt *longjmp* dies durch 1.

In einem Prozeß mit mehreren Threads (Multi-Thread) können Sie nicht mit *longjmp* den Thread wechseln. Springen Sie daher nicht zu einem *jmp_buf*, der von einem anderen Thread per *setjmp* gespeichert wurde.

Rückgabewert Keiner.

Siehe auch *ctrlbrk*, *setjmp*, *signal*

lowvideo — conio.h

Funktion Setzt das Textattribut für nachfolgende Zeichenausgaben auf »niedrige Intensität«.

Syntax

```
void lowvideo(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *lowvideo* löscht das Bit für »hohe Intensität« der aktuellen Vordergrundfarbe. Bereits ausgegebene Zeichen bleiben unverändert – es sind nur solche Zeichen betroffen, die von Funktionen mit direkter Textausgabe *nach* dem Aufruf von *lowvideo* ausgegeben werden.

➥ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *highvideo, normvideo, textattr, textcolor*

_lrotl, lrotr — stdlib.h

Funktion Rotiert einen **unsigned long**-Wert bitweise nach links oder rechts.

Syntax

```
unsigned long _lrotl(unsigned long val, int count);
unsigned long _lrotr(unsigned long val, int count);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_lrotl* rotiert den als *val* übergebenen Wert um *count* Bitpositionen nach links. *_lrotr* rotiert den als *val* übergebenen Wert um *count* Bitpositionen nach rechts.

Rückgabewert *_lrotl* liefert den rotierten Integerwert zurück:

- *_lrotl* liefert den um *count* Bitpositionen nach links rotierten Wert von *val* zurück.
- *_lrotr* liefert den um *count* Bitpositionen nach rechts rotierten Wert von *val* zurück.

Siehe auch *_crotr, _crotl, _rotl, _rotr*

K–L

lsearch — stdlib.h

Funktion Führt eine lineare Suche durch.

Syntax

```
void *lsearch(const void *key, void *base, size_t *num, size_t width,
              int (_USERENTRY *fcmp)(const void *, const void *));
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *lsearch* dient zum Durchsuchen von Tabellen. Da es sich um eine lineare Suche handelt, können die Elemente im Gegensatz zu *bsearch* auch in unsortierter Form vorliegen.

Wird das durch den Parameter *key* bezeichnete Element in der Tabelle nicht gefunden, dann wird es neu in die Tabelle aufgenommen.

- Der Parameter *base* muß auf das erste Element der Tabelle zeigen.

- *num* zeigt auf einen Integer, der die Anzahl der Tabellenelemente enthält.
- *width* gibt die Größe eines einzelnen Elements in Bytes an.
- *key* zeigt auf das zu suchende Element (den *Suchschlüssel*).

Die Funktion *fcmp* muß mit der _USERENTRY Aufrufkonvention verwendet werden.

Der Parameter *fcmp* zeigt auf eine vom Programmierer erstellte Routine, die jeweils zwei Elemente vergleicht, und einen auf diesem Vergleich basierenden Wert zurückliefert.

Während der Suche ruft *lsearch* diese Routine, deren Adresse in *fcmp* übergeben wird, mehrfach auf.

Bei jedem Aufruf der Vergleichsroutine übergibt *lsearch* dieser zwei Argumente: *key*, einen Zeiger auf den Suchschlüssel, und *elem*, einen Zeiger auf das zu vergleichende Tabellenelement.

Auf welche Art und Weise die Vergleichsroutine arbeitet, ist allein Sache des Programmierers.

Rückgabewert Die Funktion liefert die Adresse des ersten Elements der Tabelle zurück, das mit dem Suchschlüssel identisch ist. Wenn **key* ungleich **elem* ist, muß *fcmp* einen Wert ungleich null zurückliefern, ansonsten den Wert 0.

Siehe auch *bsearch*, *lfind*, *qsort*

lseek io.h

Funktion Setzt die aktuelle Position des Dateizeigers innerhalb einer Datei.

Syntax

```
long lseek(int handle, long offset, int fromwhere);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *lseek* setzt den Dateizeiger der durch *handle* bezeichneten Datei auf eine neue Position, die *offset* Bytes von der durch *fromwhere* angegebenen Dateiposition entfernt ist. *fromwhere* muß dabei einer der folgenden in io.h definierten symbolischen Konstanten sein:

fromwhere	Dateiposition
SEEK_CUR	Aktuelle Position des Dateizeigers
SEEK_END	Dateiende
SEEK_SET	Dateianfang

Rückgabewert *lseek* liefert bei fehlerfreier Ausführung die neue Zeigerposition, gemessen in Bytes ab Dateianfang, zurück. Im Fehlerfall wird der Wert -1L zu-

rückgeliefert, und die globale Variable *errno* erhält einen der folgenden Werte:

EBADF	Ungültiges Handle
EINVAL	Ungültiges Argument
ESPIPE	Suche auf Gerät nicht erlaubt

Bei der Anwendung auf Gerätedateien (wie Terminals, Druckern, Schnittstellen usw.) ist der Rückgabewert undefiniert.

Siehe auch *filelength, fseek, ftell, getc, open, _rtl_write, sopen, ungetc, write*

ltoa — stdlib.h

Funktion Konvertiert einen **long**-Wert in einen String.

Syntax

```
char *ltoa(long value, char *string, int radix);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *ltoa* konvertiert den übergebenen Wert *value* in einen nullterminierten String und speichert den Rückgabewert an der durch *string* angegebenen Adresse. *ltoa* erwartet für *value* einen **long**-Wert.

Der Parameter *radix* legt die Basis fest, auf der die Konvertierung stattfinden soll – mögliche Werte liegen im Bereich von 2 bis 36. Wenn *radix* den Wert 10 hat und *value* kleiner als null ist, stellt *ltoa* dem Ergebnis ein Minuszeichen voran.

Der für *string* reservierte Speicher muß groß genug sein, um das Ergebnis der Konvertierung inklusive des abschließenden Nullzeichens (\0) aufnehmen zu können. *ltoa* liefert bis zu 33 Bytes zurück.

Rückgabewert Die Funktion liefert den Zeiger *string* zurück.

Siehe auch *itoa, ultoa*

_makepath — stdlib.h

Funktion Erzeugt einen Pfadnamen aus einzelnen Komponenten.

Syntax

```
void _makepath(char *path, const char *drive, const char *dir,
               const char *name, const char *ext);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

_makepath erzeugt einen Pfadnamen aus Einzelkomponenten. Der neue Pfadname heißt

```
X:\DIR\SUBDIR\NAME.EXT
```

wobei gilt:

drive = X:
dir = \DIR\SUBDIR\
name = NAME
ext = .EXT

Ist *drive* leer oder NULL, wird keine Laufwerksbezeichnung in den Pfadnamen eingefügt. Fehlt nach *drive* der abschließende Doppelpunkt (:), wird er automatisch in den Pfadnamen eingefügt.

Ist *dir* leer oder NULL, wird kein Verzeichnis in den Pfadnamen eingefügt. Fehlt nach *dir* der abschließende Schrägstrich (\ oder /), wird automatisch ein umgekehrter Schrägstrich (Backslash) in den Pfadnamen eingefügt.

Ist *name* leer oder NULL, wird kein Dateiname in den Pfadnamen eingefügt.

Ist *ext* leer oder NULL, wird keine Namenserweiterung in den Pfadnamen eingefügt. Fehlt der Punkt (.) zu Beginn der Namenserweiterung, wird er automatisch in den Pfadnamen eingefügt.

_makepath setzt voraus, daß der mit *path* übergebene String eine ausreichende Größe für den konstruierten Pfadnamen hat. Die Maximalgröße ist _MAX_PATH, wobei _MAX_PATH in stdlib.h definiert ist.

_makepath und *_splitpath* verhalten sich invers zueinander. Die Anwendung von *_makepath* auf die durch *_splitpath* ermittelten Elemente eines vollständigen Pfadnamens ergeben wieder das Original.

Rückgabewert

Keiner.

Siehe auch

_fullpath, *_splitpath*

malloc — stdlib.h

Funktion

Reserviert Hauptspeicher.

Syntax

```
void *malloc(size_t size);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

malloc reserviert einen Speicherbereich von *size* Bytes auf dem Heap. Damit kann ein Programm Speicher genau nach Bedarf und exakt in der benötigten Größe reservieren.

Der Heap wird für die dynamische Reservierung von Speicherbereichen variabler Größe benutzt. Viele Datenstrukturen wie beispielsweise Bäume und Listen reservieren Speicher auf dem Heap.

In den kleinen Datenmodellen wird mit **near**-Zeigern gearbeitet, der Heap liegt zwischen dem oberen Ende des Datensegments und dem unteren Ende des Stack. Der Heap reserviert den gesamten, hier zur Verfügung stehenden Speicherbereich bis auf einige Bytes direkt unterhalb der Stack-Spitze, die für Stack-Vergrößerungen und für das Betriebssystem reserviert sind.

In den großen Datenmodellen wird mit **far**-Zeigern gearbeitet, der Heap beginnt oberhalb des Stack, und endet mit der höchsten, vom System zur Verfügung gestellten Speicheradresse.

Rückgabewert

malloc liefert bei fehlerfreier Ausführung einen Zeiger, der auf den neu reservierten Speicherbereich zeigt. Wenn kein Speicherbereich ausreichender Größe zur Verfügung steht, dann ist der Rückgabewert NULL. Es findet keine Initialisierung des Speicherblocks statt. Wenn der Parameter *size* den Wert 0 hat, liefert *malloc* NULL zurück.

Siehe auch

calloc, *farcalloc*, *farmalloc*, *free*, *realloc*

_matherr, _matherrl — math.h

Funktion

Vom Programmierer modifizierbare Routinen zur Behandlung von Berechnungsfehlern.

Syntax

```
int _matherr(struct _exception *e);
int _matherrl(struct _exceptionl *e);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

_matherr wird aufgerufen, wenn die Mathematikbibliothek einen Fehler erzeugt.

_matherrl ist die **long double** Version. Sie wird aufgerufen, wenn eine mathematische Funktion vom Typ **long double** einen Fehler erzeugt.

Sie können *_matherr* und *_matherrl* modifizieren und eigene Routinen zur Fehlerbehandlung schreiben (im Beispiel finden Sie eine entsprechende Implementierung von *_matherr*).

_matherr und *_matherrl* sind besonders zum Aufspüren von Definitions- und Wertebereichsfehlern geeignet. Besondere Fehler der Gleitkommaberechnung (wie Division durch null) werden nicht gefunden. Solche Fehler müssen mit *signal* abgefangen werden.

_matherr und *_matherrl* können vom Programmierer für die Behandlung spezieller Fehler beliebig erweitert und modifiziert werden. Anstelle der Version in der C-Bibliothek wird dann die modifizierte Funktion verwendet. Wenn die Routine den Fehler korrigieren kann, sollte sie einen Wert ungleich null zurückliefern. Falls der Fehler nicht korrigiert werden kann, muß ihr Rückgabewert der Wert 0 sein. Liefern *_matherr* oder *_matherrl* einen Wert ungleich null zurück, dann wird die globale Variable *errno* nicht geändert und es wird keine Fehlermeldung ausgegeben.

Die Strukturtypen *_exception* und *_exceptionl* sind in math.h folgendermaßen definiert:

```
struct _exception {
   int    type;        /* Fehlerart */
   char   *name;       /* Funktionsname */
   double arg1,        /* erstes Argument */
          arg2,        /* zweites Argument (wenn vorhanden) */
          retval;      /* Ergebnis */
};

struct _exceptionl {
   int    type;        /* Fehlerart */
   char   *name;       /* Funktionsname */
   long double arg1,   /* erstes Argument */
               arg2,   /* zweites Argument (wenn vorhanden) */
               retval; /* Ergebnis */
};
```

Die Felder von *_exception* und *_exceptionl* werden in der folgenden Tabelle beschrieben:

Felder	Was es ist (oder repräsentiert)
type	beschreibt die Art des aufgetretenen Fehlers und ist in math.h als Aufzählungstyp *_mexcep* definiert (siehe Liste weiter unten).
name	zeigt auf einen nullterminierten String mit dem Namen der mathematischen Funktion, durch die der Fehler ausgelöst wurde.
arg1, *arg2*	enthalten die Werte der Argumente, die der mathematischen Funktion übergeben wurden. Wenn die Funktion nur einen Wert erwartet, dann ist dieser Wert in *arg1* gespeichert.
retval	enthält den Standardwert, den *_matherr* oder *_matherrl* zurückliefern. Dieser Wert kann vom Programmierer geändert werden.

Der ebenfalls in math.h definierte Aufzählungstyp *_mexcep* beschreibt die die folgenden Fehlerarten:

Symbolische Konstante	Mathematische Fehler
DOMAIN	Das Argument liegt außerhalb des gültigen Definitionsbereichs für die Funktion. Beispiel: *log(-1)*.
SING	Der Rückgabewert wäre eine mathematische Singularität. Beispiel: *pow(0, -2)*.
OVERFLOW	Das Argument würde einen Rückgabewert erzeugen, dessen Wert größer als DBL_MAX (bzw. LDBL_MAX) ist. Beispiel: *exp(1000)*.
UNDERFLOW	Das Argument würde einen Rückgabewert erzeugen, dessen Wert kleiner als DBL_MIN (bzw. LDBL_MIN) ist. Beispiel: *exp(-1000)*.
TLOSS	das Argument würde einen Rückgabewert erzeugen, bei dem ämtliche Dezimalstellen ungenau sind. Beispiel: *sin(10e70)*.

Die Makros DBL_MAX, DBL_MIN, LDBL_MAX und LDBL_MIN sind in float.h definiert.

Den Quelltext der vordefinierten Versionen von *_matherr* und *_matherrl* finden Sie auf den Borland C++ Disketten.

Die UNIX-Versionen von *_matherr* und *_matherrl* (Ausgabe einer Meldung und Programmende) sind nicht ANSI-kompatibel. Sie finden die UNIX-Versionen dieser Routinen in den Dateien MATHERR.C und MATHERRL.C auf den Borland C++ Disketten.

Rückgabewert

Die vordefinierten Versionen von *_matherr* und *_matherrl* liefern für UNDERFLOW und TLOSS den Wert 1 zurück – in allen anderen Fällen wird der Wert 0 zurückgeliefert. *_matherr* und *_matherrl* könnenden Wert von *e->retval* beliebig verändern. *e->retval* wird dann von der Routine zurückgeliefert, die den Fehler verursacht hat.

Wenn *_matherr* oder *_matherrl* den Wert 0 zurückliefern (und damit anzeigen, daß der Fehler nicht behoben werden konnte), erhält die globale Variable *errno* den Wert 0 und eine Fehlermeldung wird ausgegeben.

Liefern *_matherr* oder *_matherrl* einen Wert ungleich null zurück, wird *errno* nicht gesetzt und keine Fehlermeldung ausgegeben.

M

max — stdlib.h

Funktion

Liefert den größeren von zwei Werten zurück.

Syntax

```
(type) max(a,b);
template <class T> T max( T t1, T t2 );
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Das C-Makro und die C++ Template-Funktion vergleichen zwei Werte *a* und *b* und liefern den größeren der beiden Werte zurück. Sowohl die beiden Argumente als auch die Deklaration der Routine müssen vom selben Typ sein.

Rückgabewert *max* liefert den größeren der beiden Werte zurück.

Siehe auch *min*

mblen stdlib.h

Funktion Ermittelt die Länge eines Multibyte-Zeichens.

Syntax

```
int mblen(const char *s, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung Ist *s* ungleich NULL, bestimmt *mblen* die Anzahl der Bytes des Zeichens, auf das *s* zeigt. Der Parameter *n* legt fest, wieviele Bytes maximal geprüft werden.

Das Verhalten von *mblen* ist von der Locale-Kategorie LC_CTYPE abhängig.

Rückgabewert *mblen* liefert einen Wert ungleich null zurück, wenn *s* den Wert NULL hat und Multibyte-Zeichen Statuscodierungen enthalten. Andernfalls liefert *mblen* den Wert 0 zurück.

Ist *s* ungleich NULL, dann liefert *mblen* den Wert 0, wenn *s* auf ein Nullzeichen (\0) zeigt. Enthalten die nächsten *n* Bytes kein gültiges Multibyte-Zeichen, so wird -1 zurückgeliefert. Ansonsten wird die Anzahl der Bytes zurückgeliefert, die ein gültiges Multibyte-Zeichen enthalten.

Siehe auch *mbstowcs*, *mbtowc*, *setlocale*

mbstowcs stdlib.h

Funktion Konvertiert einen Multibyte-String in ein *wchar_t*-Array.

Syntax

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *mbstowcs* konvertiert den Multibyte-String *s* in das durch *pwcs* bezeichnete Array. In diesem Array werden maximal *n* Werte gespeichert. Trifft

mbstowcs auf eine ungültige Multibyte-Sequenz, wird *(size_t)-1* zurückgeliefert.

Wenn *mbstowcs* den Wert *n* zurückliefert, wird das Array *pwcs* nicht durch ein Nullzeichen abgeschlossen.

Das Verhalten von *mbstowcs* ist von der Locale-Kategorie LC_CTYPE abhängig.

Rückgabewert Die Funktion *mbstowcs* liefert *(size_t)-1* zurück, wenn sie auf eine ungültige Multibyte-Sequenz trifft. Andernfalls wird die Anzahl der geänderten Array-Elemente zurückgeliefert (das abschließende Zeichen ist – falls vorhanden – darin nicht enthalten).

Siehe auch *mblen*, *mbtowc*, *setlocale*

mbtowc — stdlib.h

Funktion Konvertiert ein Multibyte-Zeichen in einen Wert des Typs *wchar_t*.

Syntax

```
int mbtowc (wchar_t *pwc, const char *s, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung Ist *s* ungleich NULL, stellt *mbtowc* die Anzahl der Bytes fest, aus der das durch *s* angegebene Multibyte-Zeichen besteht. Anschließend prüft *mbtowc* den Wert des Typs *wchar_t*, der dem Multibyte-Zeichen entspricht. Besteht eine Übereinstimmung zwischen *wchar_t* und dem Multibyte-Zeichen und ist *pwc* ungleich NULL, wird der Wert von *wchar_t* in dem Array gespeichert, auf das *pwc* zeigt. Der Parameter *n* legt fest, wieviele Bytes maximal untersucht werden.

Rückgabewert Zeigt *s* auf ein ungültiges Multibyte-Zeichen, wird -1 zurückgeliefert. Zeigt *s* auf ein Nullzeichen, hat *mbtowc* den Rückgabewert 0. In allen anderen Fällen liefert *mbtowc* die Anzahl der Bytes zurück, aus denen sich das konvertierte Multibyte-Zeichen zusammensetzt.

Der Rückgabewert ist nie größer als MB_CUR_MAX oder der Wert von *n*.

Das Verhalten von *mbstowcs* ist von der Locale-Kategorie LC_CTYPE abhängig.

Siehe auch *mblen*, *mbstowcs*, *setlocale*

memccpy, _fmemccpy — mem.h

Funktion Kopiert einen Block mit *n* Bytes.

Syntax

```
void *memccpy(void *dest, const void *src, int c, size_t n)
void far * far _fmemccpy(void far *dest, const void far *src, int c, size_t
n)
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memcpy	■	■	■	■			■
_fmemccpy	■		■				

Beschreibung *memccpy* ist unter UNIX System V verfügbar.

Die Funktion kopiert einen Block mit *n* Bytes von *src* nach *dest*. Der Kopiervorgang wird abgebrochen wenn folgendes eintritt:

- Das Zeichen *c* wird zum erstenmal nach *dest* kopiert.
- *n* Bytes wurden nach *dest* kopiert.

Rückgabewert *memccpy* liefert NULL zurück, wenn *n* Bytes kopiert wurden (und kein Byte des Wertes *c* dabei gefunden wurde). Wurde *memccpy* dagegen durch die Kopie eines Byte mit dem Wert *c* beendet, dann wird ein Zeiger zurückgeliefert, der auf das nächste Byte nach dem kopierten Wert *c* in *dest* zeigt.

Siehe auch *memcpy*, *memmove*, *memset*

memchr, _fmemchr — mem.h

Funktion Sucht *n* Bytes nach dem Wert *c* ab.

Syntax

```
void *memchr(const void *s, int c, size_t n);                  /* nur C */
void far * far _fmemchr(const void far *s, int c, size_t n); /* C und C++ */
const void *memchr(const void *s, int c, size_t n);             // nur C++
void *memchr(void *s, int c, size_t n);                         // nur C++
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memchr	■	■	■	■	■	■	■
_fmemchr	■		■				

Beschreibung *memchr* ist unter UNIX System V verfügbar.

Die Funktion sucht die ersten *n* Bytes des Blocks *s* nach dem Wert *c* ab.

Rückgabewert

memchr liefert die Adresse des ersten Vorkommens von *c* im Array *s* zurück – oder NULL, wenn *c* innerhalb der ersten *n* Bytes von *s* nicht vorkommt.

➥ Wenn Sie die Intrinsic-Version der Funktionen benutzen, wird für *n=0* der Wert NULL zurückgeliefert.

memcmp, _fmemcmp — mem.h

Funktion

Vergleicht die ersten *n* Bytes zweier Blöcke.

Syntax

```
int memcmp(const void *s1, const void *s2, size_t n);
int far * far _fmemcmp(const void far *s1, const void far *s2, size_t n);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memcmp	■	■	■	■	■	■	■
_fmemcmp	■		■				

Beschreibung

memcmp ist unter UNIX System V verfügbar.

Die Funktion vergleicht die ersten *n* Bytes der Blöcke *s1* und *s2* miteinander. *memcmp* behandelt die Bytes als **unsigned char**.

Rückgabewert

Da die Bytes als **unsigned char** verglichen werden, liefert *memcmp* einen der folgenden Werte zurück:

- < 0 falls *s1* kleiner als *s2* ist
- == 0 falls die ersten *n* Zeichen von *s1* und *s2* übereinstimmen
- \> 0 falls *s1* größer als *s2* ist

Beispielsweise liefert

```
memcmp("\xFF", "x7F", 1)
```

einen Wert größer null.

➥ Wenn Sie die Intrinsic-Version der Funktionen benutzen, wird für *n=0* der Wert 0 zurückgeliefert.

Siehe auch

memicmp

memcpy, _fmemcpy — mem.h

Funktion

Kopiert *n* Bytes von *src* nach *dest*.

M

Syntax

```
void *memcpy(void *dest, const void *src, size_t n);
void far * far _fmemcpy(void far *dest, const void far *src, size_t n);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memcpy	■	■	■	■	■	■	■
_fmemcpy	■		■				

Beschreibung

memcpy ist unter UNIX System V verfügbar.

Die Funktion kopiert *n* Bytes von *src* nach *dest*. Überlappen sich *scr* und *dest*, ist das Verhalten von *memcpy* undefiniert.

Rückgabewert

memcpy liefert den übergebenen Parameter *dest* zurück.

Siehe auch

memccpy, *memmove*, *memset*, *movedata*, *movmem*

memicmp, _fmemicmp — mem.h

Funktion

Vergleicht die ersten *n* Bytes zweier Arrays, wobei zwischen Groß- und Kleinschreibung nicht unterschieden wird.

Syntax

```
int memicmp(const void *s1, const void *s2, size_t n);
int far _fmemicmp(const void far *s1, const void far *s2, size_t n);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memicmp	■	■	■	■			■
_fmemicmp	■		■				

Beschreibung

memicmp ist unter UNIX System V verfügbar.

Die Funktion vergleicht die ersten *n* Bytes der Blöcke *s1* und *s2* miteinander, wobei zwischen Groß- und Kleinschreibung nicht unterschieden wird.

Rückgabewert

memicmp bricht den Vergleich beim ersten nicht übereinstimmenden Zeichen ab. Das Vergleichsergebnis wird als **int** zurückgeliefert:

- < 0 falls *s1* kleiner als *s2* ist
- == 0 falls die ersten *n* Zeichen von *s1* und *s2* übereinstimmen
- > 0 falls *s1* größer als *s2* ist

Siehe auch

memcmp

memmove, _fmemmove — mem.h

Funktion

Kopiert einen Block von *n* Bytes.

Syntax
```
void *memmove(void *dest, const void *src, size_t n);
void far * far _fmemmove(void far *dest, const void far *src, size_t n);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memmove	■	■	■	■	■	■	■
_fmemmove	■		■				

Beschreibung Die Funktion kopiert *n* Bytes von *src* nach *dest* und ist daher im Prinzip identisch mit *memcpy* – hier werden jedoch Überlappungen von Quell- und Zielgebiet automatisch berücksichtigt.

Rückgabewert *memmove* liefert den übergebenen Parameter *dest* zurück, also einen Zeiger auf den Zielblock.

Siehe auch *memccpy, memcpy, movmem*

memset, _fmemset — mem.h

Funktion Setzt die ersten *n* Bytes des Speicherblocks *s* auf den Wert von *c*.

Syntax
```
void *memset(void *s, int c, size_t n);
void far * far _fmemset(void far *s, int c, size_t n);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
memset	■	■	■	■	■	■	■
_fmemset	■		■				

M

Beschreibung *memset* setzt die ersten *n* Bytes des Array *s* auf den Wert von *c*.

Rückgabewert *memset* liefert den übergebenen Parameter *s* zurück.

Siehe auch *memccpy, memcpy, setmem*

min — stdlib.h

Funktion Liefert den kleineren von zwei Werten zurück.

Syntax
```
(type) min(a,b);
template ass T T min( T t1, T t2 );
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Das C-Makro und die C++ Template-Funktion vergleichen zwei Werte *a* und *b* und liefern den kleineren der beiden Werte zurück. Sowohl die bei-

den Argumente als auch die Deklaration der Routine müssen vom selben Typ sein.

Rückgabewert *min* liefert den kleineren der beiden Werte zurück.

Siehe auch *max*

mkdir dir.h

Funktion Erzeugt ein Verzeichnis.

Syntax

```
int mkdir(const char *path);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *mkdir* ist unter UNIX verfügbar, benötigt dort aber einen zusätzlichen *filemode*-Parameter.

mkdir erzeugt ein Verzeichnis mit dem in *path* angegebenen Namen.

Rückgabewert *mkdir* liefert den Wert 0 zurück, wenn das neue Verzeichnis erzeugt werden konnte.

Wenn ein Fehler auftritt, so ist der Rückgabewert -1 und die globale Variable *errno* wird auf einen der beiden folgenden Werte gesetzt:

EACCES	Zugriff nicht gestattet
ENOENT	Datei oder Verzeichnis nicht gefunden

Siehe auch *chdir*, *getcurdir*, *getcwd*, *rmdir*

MK_FP dos.h

Funktion Erzeugt einen **far**-Zeiger.

Syntax

```
void far *MK_FP(unsigned seg, unsigned ofs);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *MK_FP* ist ein Makro, über das ein **far**-Zeiger aus seinen Komponenten *seg* (Segment) und *ofs* (Offset) erzeugt werden kann.

Rückgabewert *MK_FP* liefert einen **far**-Zeiger zurück.

Siehe auch *FP_OFF*, *FP_SEG*, *movedata*, *segread*

mktemp — dir.h

Funktion Erzeugt einen eindeutigen Dateinamen.

Syntax

```
char *mktemp(char *template);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *mktemp* ersetzt den durch *template* bezeichneten String durch einen eindeutigen Dateinamen und liefert diesen zurück.

template sollte ein nullterminierter String mit sechs abschließenden X-Zeichen sein. Diese X-Zeichen werden, beginnend mit *AA.AAA*, durch zwei Buchstaben, einen Punkt und drei Buchstaben für die Dateinamenserweiterung ersetzt.

mktemp verwandelt *XXXXXX* als erstes in *AA.AAA* (macht also aus *tdatXXXXXX* den Namen *tdatAA.AAA*) und prüft dann innerhalb des angegebenen bzw. aktuellen Verzeichnisses, ob eine Datei dieses Namens bereits existiert. Wenn ja, wird der Versuch mit *AA.AAB*, *AA.AAC* usw. solange wiederholt, bis sich der Name einer noch nicht existierenden Datei ergibt.

Rückgabewert Falls *template* richtig aufgebaut ist, liefert *mktemp* die Adresse des Strings *template* zurück. Andernfalls wird NULL zurückgeliefert.

mktime — time.h

Funktion Konvertiert Uhrzeit in das Kalenderformat.

Syntax

```
time_t mktime(struct tm *t);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *mktime* konvertiert die Uhrzeit in der Struktur, auf die *t* zeigt, in eine Kalenderzeit mit dem Format der Funktion *time*. Die ursprünglichen Werte der Felder *tm_sec*, *tm_min*, *tm_hour*, *tm_mday* und *tm_mon* sind nicht auf den durch die *tm*-Struktur festgelegten Bereich beschränkt. Befinden sich die Felder nicht im korrekten Bereich, werden sie angepaßt. Die Werte der Felder *tm_wday* und *tm_yday* werden nach der Anpassung der anderen Felder berechnet. Kann die Kalenderzeit nicht dargestellt werden, liefert *mktime* -1 zurück.

Die zulässige Bereich für die Kalenderzeit liegt zwischen dem 1. Januar 1970 00:00:00 und dem 19. Januar 2038 03:14:07.

Rückgabewert Siehe Beschreibung.

Siehe auch *localtime, strftime, time*

modf, modfl — math.h

Funktion Teilt einen **double-** oder **long double**-Wert in Vor- und Nachkommaanteil auf.

Syntax

```
double modf(double x, double *ipart);
long double modfl(long double x, long double *ipart);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
	■	■	■	■	■	■	■
modfl	■	■	■	■	■	■	■

Beschreibung *modf* spaltet den **double**-Wert *x* in zwei Teile: in den Vorkomma- und den Nachkommaanteil. *modf* speichert den ganzzahligen Vorkommaanteil in *ipart* und liefert den Nachkommaanteil zurück.

modfl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert *modf* und *modfl* liefern den Nachkommaanteil des Arguments *x* zurück.

Siehe auch *fmod, ldexp*

movedata — mem.h

Funktion Kopiert *n* Bytes.

Syntax

```
void movedata(unsigned srcseg, unsigned srcoff, unsigned dstseg,
              unsigned dstoff, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *movedata* kopiert *n* Bytes von der Quelladresse (*srcseg: srcoff*) zur Zieladresse (*dstseg:dstoff*). *movedata* funktioniert unabhängig vom verwendeten Speichermodell.

Rückgabewert Keiner.

Siehe auch *FP_OFF, memcpy, MK_FP, movmem, segread*

movetext — conio.h

Funktion

Kopiert einen rechteckigen Ausschnitt des Textbildschirms in einen anderen Ausschnitt.

Syntax

```
int movetext(int left, int top, int right, int bottom, int destleft, int desttop);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

movetext kopiert einen rechteckigen Ausschnitt des Textbildschirms, dessen obere linke Ecke durch (*left*, *top*) und dessen untere rechte Ecke durch (*right*, *bottom*) festgelegt ist, in einen rechteckigen Ausschnitt gleicher Größe, dessen obere linke Ecke mit (*destleft*, *desttop*) angegeben wird.

movetext arbeitet mit absoluten Bildschirm-Koordinaten, d.h. unabhängig von einem eventuell gesetzten Textfenster. Überlappende Ausschnitte werden korrekt kopiert.

movetext funktioniert nur im Textmodus und schreibt direkt in den Bildschirmspeicher.

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert

Bei fehlerfreier Ausführung liefert *movetext* einen Wert ungleich null zurück. Wenn *movetext* nicht ausgeführt werden konnte, weil entweder der Quellausschnitt oder der Zielausschnitt außerhalb des Bereichs des aktuellen Bildschirmmodus liegen, ist der Rückgabewert 0.

Siehe auch

gettext, *puttext*

movmem, _fmovmem — mem.h

Funktion

Verschiebt einen Block der Länge *length* Bytes.

Syntax

```
void movmem(const void *src, void *dest, unsigned length);
void _fmovmem(void far *src, void far *dest, unsigned length);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

movmem verschiebt *length* Bytes von *src* nach *dest*. Quell- und Zielbereich können sich überlappen – *movmem* verschiebt die Daten in jedem Fall korrekt. *_fmovmem* bietet für das Speichermodell Large dieselbe Funktionalität wie *movmem* für die kleinen Speichermodelle.

Rückgabewert

Keiner.

Siehe auch *memcpy*, *memmove*, *movedata*

_msize — malloc.h

Funktion Liefert die Größe eines Heap-Blocks zurück.

Syntax

```
void _msize(void *block);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung *_msize* liefert die Größe des reservierten Heap-Blocks mit der Adresse *block* zurück. Der Block muß zuvor mit *malloc*, *calloc* oder *realloc* reserviert worden sein. Der zurückgelieferte Wert kann größer sein als der, der bei der Speicherreservierung angefordert wurde.

Rückgabewert *_msize* liefert die Blockgröße in Bytes zurück.

Siehe auch *malloc*, *free*, *realloc*

normvideo — conio.h

Funktion Setzt normale Intensität für nachfolgende Textausgaben.

Syntax

```
void normvideo(void);
```

DOS	UNIX	Windows	ANSI C	nur C++	
■			■		■

Beschreibung *normvideo* setzt das Textattribut (Vorder- und Hintergrund), das beim Start des Programms gültig war.

Bereits *ausgegebene* Zeichen bleiben unverändert – es wird lediglich das Zeichenattribut für Funktionen wie z.B. *cprintf* festgelegt, die direkte Bildschirmausgaben durchführen, *nachdem* die Funktion *normvideo* aufgerufen wurde.

Rückgabewert Keiner.

Siehe auch *highvideo*, *lowvideo*, *textattr*, *textcolor*

offsetof — stddef.h

Funktion Ermittelt den Offset eines Strukturelementes in Bytes.

Syntax

```
size_t offsetof(struct_type, struct_member);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

offsetof ist nur als Makro verfügbar. Der Parameter *struct_tpye* ist ein **struct**-Typ. *struct_member* ist ein beliebiges Element dieser Struktur, das über den Elementauswahloperator oder über Zeiger angesprochen werden kann.

Wird für *struct_member* ein Bitfeld angegeben, so ist das Ergebnis undefiniert.

Im Kapitel 2 des *Programmierhandbuchs* finden Sie Informationen über den Operator **sizeof** und über Speicherreservierung und Ausrichtung von Strukturen.

Rückgabewert

offsetof liefert die Anzahl der Bytes zurück, die zwischen der Startadresse der Struktur und der Adresse des Strukturelements liegen.

_open — fcntl.h, share.h, dos.h

Beschreibung

Veraltete Funktion. Siehe *_rtl_open*.

open — fcntl.h, io.h

Funktion

Öffnet eine Datei für Lese- oder Schreibaktionen.

Syntax

```
int open(const char *path, int access[, unsigned mode]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

O

Beschreibung

open öffnet die durch *path* angegebene Datei und bereitet sie je nach Inhalt des Parameters *access* für Lese- und/oder Schreibzugriffe vor.

Um beim Erzeugen einer Datei einen bestimmten Zugriffsmodus festzulegen, kann die globale Variable *_fmode* verwendet oder ein Aufruf von *open* mit den Optionen O_CREAT und O_TRUNC und dem gewünschten Übersetzungssmodus (O_BINARY bzw. O_TEXT) durchgeführt werden. Beispielsweise erzeugt der Aufruf

```
open("XMP", O_CREAT|O_TRUNC|O_BINARY,S_IREAD)
```

die Datei XMP im binären Modus für einen Nur-Lesen-Zugriff und kürzt Sie – falls sie bereits existiert – auf 0 Bytes.

Für den Parameter *access* sind zwei Arten von Flag-Konstanten in fcntl.h definiert, die über eine ODER-Operation miteinander verbunden werden müssen. Aus der ersten der beiden folgenden Listen muß genau ein Flag verwendet werden. Die restlichen Flags können in jeder logischen Kombination verwendet werden.

Diese symbolischen Konstanten sind in fcntl.h definiert.

Liste 1: Flags für Lesen und/oder Schreiben

O_RDONLY	Nur Lesen
O_WRONLY	Nur Schreiben
O_RDWR	Lesen und Schreiben

Liste 2: Weitere Zugriff-Flags

O_NDELAY	Nur aus Gründen der UNIX-Kompatibilität definiert, unter DOS nicht verwendet.
O_APPEND	Neu geschriebene Daten werden grundsätzlich angehängt, das Überschreiben bereits existierender Daten ist nicht möglich.
O_CREAT	Wenn die Datei noch nicht existiert, wird sie mit den über mode angegebenen Attributen erzeugt. Falls die Datei bereits existiert, hat dieses Flag keine Wirkung.
O_TRUNC	Wenn die Datei bereits existiert, wird ihr vorheriger Inhalt gelöscht, die Dateiattribute bleiben unverändert.
O_EXCL	Nur zusammen mit O_CREAT. Falls die Datei bereits existiert, wird ein Fehler zurückgeliefert.
O_BINARY	Die Datei wird unabhängig von *_fmode* im binären Modus geöffnet – gelesene bzw. geschriebene Daten werden nicht interpretiert.
O_TEXT	Die Datei wird unabhängig von *_fmode* im Textmodus geöffnet.

Wenn Sie weder O_BINARY noch O_TEXT angeben, wird die Datei in dem in *_fmode* gesetzten Modus geöffnet.

Bei der Anwendung von O_CREAT (d.h. der Erzeugung einer neuen Datei) muß der Parameter *mode* über die folgenden symbolischen Konstanten festgelegt werden, die in sys/stat.h definiert sind:

Wert von *mode*	Zugriffserlaubnis
S_IWRITE	Schreiben erlaubt
S_IREAD	Lesen erlaubt
S_IREAD \| S_IWRITE	Lesen und Schreiben erlaubt

Rückgabewert

Wenn die Datei fehlerfrei geöffnet werden konnte, liefert *open* das *Handle* dieser Datei zurück (einen Integerwert größer 0, über den weitere Operationen mit dieser Datei ausgeführt werden). Der Dateizeiger – er markiert die aktuelle Position in der Datei – wird auf den Dateianfang gesetzt.

Im Fehlerfall liefert die Funktion den Wert -1 zurück und setzt die globale Variable *errno* auf einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
EINVACC	Ungültiger Zugriffscode
EMFILE	Zu viele Dateien offen
ENOENT	Pfad/Datei nicht gefunden

Siehe auch *chmod*, *chsize*, *close*, *creat*, *creatnew*, *createtemp*, *dup*, *dup2*, *fdopen*, *filelength*, *fopen*, *freopen*, *getftime*, *lock*, *lseek*, *read*, *_rtl_creat*, *_rtl_open*, *_rtl_write*, *sopen*, *write*

opendir — dirent.h

Funktion Öffnet einen Verzeichnis-Stream für Leseoperationen.

Syntax

```
DIR *opendir(char *dirname);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *opendir* ist auf unter UNIX auf POSIX-Systemen verfügbar.

Die Funktion öffnet den Verzeichnis-Stream *dirname* für Leseoperationen. Der Stream wird so gesetzt, daß der erste Eintrag des Verzeichnisses gelesen werden kann.

Ein Verzeichnis-Stream wird durch die in dirent.h definierte *DIR*-Struktur repräsentiert. Der Programmierer hat keinen Zugriff auf die Felder dieser Struktur. Es können gleichzeitig mehrere Verzeichnis-Streams geöffnet und gelesen werden. Während des Lesens können Verzeichniseinträge erstellt oder gelöscht werden.

Mit der Funktion *readdir* können aufeinanderfolgende Einträge aus einem Verzeichnis-Stream gelesen werden. Das Entfernen eines nicht mehr benötigten Verzeichnis-Stream erfolgt mit *closedir*.

Rückgabewert Bei fehlerfreier Ausführung liefert *opendir* einen Zeiger auf den Verzeichnis-Stream zurück, der in Aufrufen von *readdir*, *rewinddir* und *closedir* verwendet werden kann. Kann das Verzeichnis nicht geöffnet werden, liefert *opendir* den Wert NULL und setzt die globale Variable *errno* auf einen der folgenden Werte:

ENOENT	Das Verzeichnis existiert nicht.
ENOMEM	Nicht genug Speicher für die Reservierung eines DIR-Objekts.

Siehe auch *closedir*, *readdir*, *rewinddir*

O

_open_osfhandle io.h

Funktion Verbindet Datei-Handles.

Syntax

```
int _open_osfhandle(long osfhandle, int flags);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			

Beschreibung Die Routine *_open_osfhandle* erzeugt ein Laufzeit-Datei-Handle und verbindet es mit dem durch *osfhandle* angegebenen Datei-Handle des Betriebssystems. *flags* ist eine (bitweise) ODER-Kombination der folgenden in fcntl.h definierten Konstanten:

_O_APPEND	Setzt den Dateizeiger vor jeder Schreiboperation an das Dateiende.
_O_RDONLY	Datei wird nur für Leseoperationen geöffnet.
_O_TEXT_	Datei wird im Übersetzungsmodus »Text« geöffnet.

Rückgabewert Bei fehlerfreier Ausführung liefert *_open_osfhandle* ein Laufzeit-Datei-Handle zurück. Andernfalls wird der Wert -1 zurückgeliefert.

Siehe auch *_get_osfhandle, open*

outp conio.h

Funktion Schreibt ein einzelnes Byte an eine I/O-Adresse

Syntax

```
int outp(unsigned portid, int value);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung Das Makro *outp* schreibt das niederwertige Byte von *value* an die von *portid* angegebene Adresse.

outp ist als Inline-Makro definiert – der Compiler verwendet nur dann echte Funktionsaufrufe, wenn conio.h nicht aufgenommen wird (oder ein entsprechendes `#undef` eingefügt wird).

Rückgabewert *outp* liefert *value* zurück.

Siehe auch *inp, inpw, outpw*

outport, outportb — dos.h

Funktion
Schreibt ein Wort bzw. ein Byte an eine I/O-Adresse.

Syntax

```
void outport(int portid, int value);
void outportb(int portid, unsigned char value);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung
outport arbeitet wie die 80x86-Anweisung *OUT*: Das niederwertige Byte des durch *value* gegebenen Worts wird nach *portid* und das höherwertige Byte nach *portid + 1* ausgegeben.

Das Makro *outportb* schreibt das durch *value* gegebene Byte an die durch *portid* angegebene Adresse.

outportb ist als Inline-Makro definiert – der Compiler verwendet nur dann echte Funktionsaufrufe, wenn dos.h nicht aufgenommen oder ein entsprechendes `#undef` eingefügt wird.

Rückgabewert
Keiner.

Siehe auch
inport, inportb

outpw — conio.h

Funktion
Schreibt ein Wort an eine I/O-Adresse.

Syntax

```
unsigned outpw(unsigned portid, unsigned value);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

O

Beschreibung
Das Makro *outpw* schreibt das durch *value* gegebene 16-Bit Wort an die von *portid* bezeichnete Adresse. Über eine einzelne 16-Bit *OUT*-Anweisung wird das niederwertige Byte von *value* zu *portid* und das höherwertige Byte zu *portid* + 1 ausgegeben.

outpw ist als Inline-Makro definiert – der Compiler verwendet nur dann echte Funktionsaufrufe, wenn conio.h nicht aufgenommen oder ein entsprechendes `#undef` eingefügt wird.

Rückgabewert
outpw liefert *value* zurück.

Siehe auch
inp, inpw, outp

parsfnm dos.h

Funktion Analysiert einen Dateinamen.

Syntax

```
char *parsfnm(const char *cmdline, struct fcb *fcb, int opt);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *parsfnm* durchsucht den durch *cmdline* angegebenen String nach einem Dateinamen. Der String enthält normalerweise eine Kommandozeile. Der Dateiname wird in einem durch *fcb* angegebenen File Control Block (FCB) als Laufwerksbezeichnung, Dateiname und Erweiterung gespeichert.

Der Parameter *opt* entspricht dem Register AL in der zugehörigen DOS-Funktion 0x29. In den DOS-Referenzhandbüchern finden Sie weitere Informationen über die Analyse des Dateinamens.

Rückgabewert Wenn die Analyse fehlerfrei ausgeführt werden konnte, dann liefert *parsfnm* einen Zeiger auf das erste Zeichen von *cmdline* zurück, das nicht mehr zum Dateinamen gehört. Im Falle eines Fehlers ist der Rückgabewert NULL.

_pclose stdio.h

Funktion Wartet auf die Beendigung eines über eine Pipe gestarteten Befehls.

Syntax

```
int _pclose(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung Diese Funktion ist in Win32s Programmen nicht verfügbar.

_pclose schließt den Pipe-Stream, der durch einen vorherigen Aufruf von *_popen* erzeugt wurde, und wartet auf das Ende des zugehörigen Befehls.

Rückgabewert Bei erfolgreicher Ausführung liefert *_pclose* den Status des Kindprozesses zurück. Dies entspricht dem von *cwait* zurückgelieferten Wert mit der Ausnahme, daß das niederwertige und das höherwertige Byte des niederwertigen Worts ausgetauscht werden. Im Fehlerfall liefert *_pclose* den Wert -1 zurück.

Siehe auch *_pipe, _popen*

peek dos.h

Funktion Liest ein Wort an der Position *segment:offset*.

Syntax

```
int peek(unsigned segment, unsigned offset);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *peek* liest das Wort an der Speicheradresse *segment:offset*.

Die Funktion ist in dos.h als Inline-Makro definiert – nur wenn Datei nicht aufgenommen wurde (bzw. nach einem entsprechenden `#undef` *peek*), erzeugt der Compiler echte Funktionsaufrufe.

Rückgabewert *peek* liefert das Wort an der durch *segment:offset* angegebenen Adresse zurück.

Siehe auch *peekb*, *poke*

peekb dos.h

Funktion Liest ein Byte an der Position *segment:offset*.

Syntax

```
char peekb(unsigned segment, unsigned offset);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *peekb* liest das Byte an der Speicheradresse *segment:offset*.

Die Funktion ist in dos.h als Inline-Makro definiert – nur wenn diese Datei nicht aufgenommen wurde (bzw. nach einem entsprechenden `#undef` *peek*b), erzeugt der Compiler echte Funktionsaufrufe.

Rückgabewert *peekb* liefert den Inhalt des durch *segment:offset* adressierten Byte als **char** zurück.

Siehe auch *peek*, *pokeb*

P

perror stdio.h

Funktion Schreibt eine Systemfehlermeldung in *stderr*.

Syntax

```
void perror(const char *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■	■	■	■

Beschreibung

perror schreibt die Systemfehlermeldung der letzten Bibliotheksroutine, die die globale Variable *errno* gesetzt hat, in *stderr* (normalerweise der Bildschirm).

Zuerst wird das Argument *s* ausgegeben, dann folgt ein Doppelpunkt und die zum aktuellen Inhalt von *errno* gehörende Meldung. Zuletzt wird ein Zeilenvorschub ausgegeben. Per Konvention wird der Programm-Dateiname als Parameter *s* übergeben.

Das Array *_sys_errlist* enthält sämtliche Fehlermeldungen des Systems in der Reihenfolge aufsteigender Fehlernummern. *perror* verwendet den aktuellen Wert der globalen Variablen *errno* zur Adressierung. Keiner der Strings enthält ein Zeilenvorschubzeichen (\n).

Die globale Variable *_sys_nerr* enthält die Anzahl der Einträge im Array *_sys_errlist*.

Weitere Details zu *errno*, *_sys_errlist* und *_sys_nerr* finden Sie in Kapitel 4.

Folgende Meldungen werden von *perror* erzeugt:

Tabelle 3.2: Diese Meldungen werden unter Win 16 und Win 32 erzeugt.

Meldungen unter Win 16 und Win 32	
Arg list too big	Is a directory
Attempted to remove current directory	Math argument
	Memory arena trashed
Bad address	Name too long
Bad file number	No child processes
Block device required	No more files
Broken pipe	No space left on device
Cross-device link	No such device
Error 0	No such device or address
Exec format error	No such file or directory
Executable file in use	No such process
File already exists	Not a directory
File too large	Not enough memory
Illegal seek	Not same device
Inappropriate I/O control operation	Operation not permitted
	Path not found
Input/output error	Permission denied
Interrupted function call	Possible deadlock
Invalid access code	Read-only file system
Invalid argument	Resource busy
Invalid data	Resource temporarily unavailable
Invalid environment	Result too large
Invalid format	Too many links
Invalid function number	Too many open files
Invalid memory block address	

Tabelle 3.3
Diese Meldungen werden nur unter Win 16 und Win 32 erzeugt.

Meldungen nur unter Win 32	
Bad address	No child processes
Block device required	No space left on device
Broken pipe	No such device or address
Executable file in use	No such process
File too large	Not a directory
Illegal seek	Operation not permitted
Inappropriate I/O control operation	Possible deadlock
Input/output error	Read-only file system
Interrupted function call	Resource busy
Is a directory	Resource temporarily unavailable
Name too long	Too many links

In Win32 oder Win32s GUI-Anwendungen muß *stderr* umgelenkt werden.

Rückgabewert Keiner.

Siehe auch *clearerr*, *eof*, *freopen_strerror*, *strerror*

_pipe — fcntl.h, io.h

Funktion Erzeugt eine Pipe für Lese-/Schreiboperationen.

Syntax

```
int _pipe(int *handles, unsigned int size, int mode);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung Diese Funktion ist in Win32s Programmen nicht verfügbar.

Die Funktion *_pipe* erzeugt eine unbenannte Pipe für den Informationsaustausch zwischen Prozessen. Die Pipe wird sowohl zum Schreiben als auch zum Lesen geöffnet. Wie bei einer Datei auf Diskette/Festplatte kann eine Pipe gelesen und beschrieben werden. Sie hat jedoch keinen Namen und ihr ist auch kein permanenter Speicher zugeordnet. Die über eine Pipe ausgetauschten Daten existieren nur in einem Speicherpuffer, der vom Betriebssystem verwaltet wird.

Das Lese-Handle wird in *handles*[0], und das Schreib-Handle in *handles*[1] zurückgegeben. Das Programm kann diese Handle bei folgenden Aufrufen von *read*, *write*, *dup*, *dup2* oder *close* verwenden. Die Pipe wird gelöscht, wenn alle zu ihr gehörenden Handles geschlossen sind.

Der Parameter *size* bestimmt die Größe des internen Puffers. Der empfohlene Minimalwert beträgt 512 Bytes.

P

Der Übersetzungsmodus wird durch den Parameter *mode* wie folgt angegeben:

O_BINARY	Die Pipe wird im Binärmodus geöffnet.
O_TEXT	Die Pipe wird im Textmodus geöffnet.

Ist *mode* gleich null, dann wird der Übersetzungsmodus durch die globale Variable *_fmode* bestimmt.

Rückgabewert Bei erfolgreicher Ausführung liefert *_pipe* den Wert 0 zurück und übergibt die Pipe-Handles in *handles*[0] und *handles*[1]. Andernfalls wird -1 zurückgeliefert und die globale Variable *errno* erhält einen der folgenden Werte:

EMFILE	Zu viele offene Dateien.
ENOMEM	Nicht genug Platz im Hauptspeicher.

Siehe auch *_pclose*, *_popen*

poke — dos.h

Funktion Schreibt einen **int**-Wert an die durch *segment:offset* angegebene Speicherposition.

Syntax

```
void poke(unsigned segment, unsigned offset, int value);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *poke* speichert den durch *value* angegebenen Integer an der Adresse segment:offset.

Die Funktion ist in dos.h als Inline-Makro definiert. Nur wenn diese Datei nicht mit aufgenommen wurde (bzw. nach einem entsprechenden `#undef` *poke*), verwendet der Compiler echte Funktionsaufrufe.

Rückgabewert Keiner.

Siehe auch *peek*, *pokeb*

pokeb — dos.h

Funktion Speichert einen Byte-Wert an der Position *segment:offset*.

Syntax

```
void pokeb(unsigned segment, unsigned offset, char value);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *pokeb* schreibt den Byte-Wert *value* an die Speicherposition mit der Adresse *segment:offset*.

Die Funktion ist in dos.h als Inline-Makro definiert. Nur wenn diese Datei nicht mit aufgenommen wurde (bzw. nach einem entsprechenden `#undef` *pokeb*), verwendet der Compiler »echte« Funktionsaufrufe.

Rückgabewert Keiner.

Siehe auch *peekb, poke*

poly, polyl — math.h

Funktion Erzeugt ein Polynom aus den angegebenen Argumenten.

Syntax

```
double poly(double x, int degree, double coeffs[]);
long double polyl(long double x, int degree, long double coeffs[]);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
poly	■	■	■	■			■
polyl	■		■	■			■

Beschreibung *poly* bildet ein Polynom mit dem Grad *degree* über den Wert von *x* zusammen mit den Koeffizienten *coeffs[degree], coeffs[degree-1], ..., coeffs[0]*.

Ein Aufruf mit dem Wert 4 für *degree* berechnet beispielsweise den folgenden Ausdruck:

$$coeffs[4]x^4 + coeffs[3]x^3 + coeffs[2]x^2 + coeffs[1]x + coeffs[0]$$

polyl ist die **long double** Version. Sie erwartet Argumente vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Rückgabewert Zurückgeliefert wird das Ergebnis der Polynom-Berechnung für den gegebenen Wert *x*.

P

_popen — stdio.h

Funktion Erzeugt eine Pipe zum Kommandozeileninterpreter.

Syntax

```
FILE *_popen (const char *command, const char *mode);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung Diese Funktion ist in Win32s Programmen nicht verfügbar.

_popen erzeugt eine Pipe zum Kommandozeileninterpreter. Der Kommandozeileninterpreter wird asynchron ausgeführt. Übergeben wird die durch *command* angegebene Kommandozeile. Der String *mode* legt fest, ob die Pipe mit der Standardeingabe oder der Standardausgabe des Kommandozeileninterpreters verbunden werden soll und ob sie im Binär- oder im Textmodus geöffnet werden soll.

Der String *mode* kann einen der folgenden Werte enthalten:

Wert	Beschreibung
rt	Lesen aus Standardausgabe des Befehls (Textmodus)
rb	Lesen aus Standardausgabe des Befehls (Binärmodus)
wt	Schreiben in Standardeingabe des Befehls (Textmodus)
wb	Schreiben in Standardeingabe des Befehls (Binärmodus)

Das abschließende *t* bzw. *b* ist optional. Wird es weggelassen, so bestimmt die globale Variable *_fmode* den Übersetzungsmodus.

Mit Hilfe der Funktion *_pclose* können Sie die Pipe schließen und erhalten den vom ausgeführten Befehl zurückgelieferten Wert.

Rückgabewert

Bei fehlerfreier Ausführung liefert *_popen* einen FILE-Zeiger, mit dem entsprechend dem Inhalt von *mode* aus der Standardeingabe des Befehls gelesen bzw. in die Standardausgabe geschrieben werden kann. Im Fehlerfall liefert *_popen* NULL zurück.

Siehe auch

_pclose, *_pipe*

pow, powl — math.h

Funktion

Berechnet x^y.

Syntax

```
double pow(double x, double y);
long double powl(long double x, long double y);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
pow	■	■	■	■	■	■	■
powl	■		■	■			■

Beschreibung

pow liefert das Ergebnis der Berechnung x^y zurück.

powl ist die **long double** Version. Sie erwartet Argumente vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert

Bei fehlerfreier Ausführung liefern *pow* und *powl* das Ergebnis der Berechnung x^y zurück.

Falls sich die Argumente nicht innerhalb der zulässigen Grenzen halten, so wird ein Überlauf erzeugt oder das Ergebnis ist nicht berechenbar. Bei einem Überlauf liefert *pow* HUGE_VAL und *powl* _LHUGE_VAL. Die globale Variable *errno* wird in diesen Fällen auf folgenden Wert gesetzt:

ERANGE Ergebnis außerhalb des Rechenbereichs

Ist der Parameter *x* reellwertig und kleiner als 0, und ist *y* keine ganze Zahl (oder beim Aufruf von *pow(0,0)*, so erhält die globale Variable *errno* den Wert:

EDOM Bereichsfehler

Die Fehlerbehandlung kann für diese Funktionen über die Routinen _matherr und *_matherrl* modifiziert werden.

Siehe auch

bcd, *complex*, *exp*, *pow10*, *sqrt*

pow10, pow10l — math.h

Funktion

Berechnet 10^p.

Syntax

```
double pow10(int p);
long double pow10l(int p);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
pow10	■	■	■	■			■
pow10l	■		■	■			■

Beschreibung

pow10 berechnet den Ausdruck 10^p.

Rückgabewert

pow10 liefert das Ergebnis der Berechnung von 10^p zurück.

Das Resultat wird mit der Genauigkeit eines **long double** berechnet. Alle Argumente sind zulässig, können jedoch in manchen Fällen zu einem Überlauf oder Unterlauf führen.

pow10l ist die **long double** Version und liefert ein Ergebnis vom Typ **long double** zurück.

Siehe auch

exp, *pow*

P

printf — stdio.h

Funktion Schreibt formatiert in *stdout*.

Syntax

```
int printf(const char *format[argument, ...]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■	■	■	■

Beschreibung

printf übernimmt eine Reihe von Argumenten, auf die jeweils eine Formatierungsanweisung angewendet wird. Die Formatierungsanweisungen sind in dem String enthalten, auf den *format* zeigt. Die so formatierten Daten werden in den Stream *stdout* geschrieben. Die Anzahl der übergebenen Argumente muß mit der der Formatierungsanweisungen übereinstimmen.

In Win32 oder Win32s GUI-Anwendungen muß *stdout* umgelenkt werden.

Der Formatstring

Der Formatstring muß in jedem Aufruf einer *...printf* Funktion enthalten sein – über ihn wird die Art der Konvertierung, die Formatierung und die Anzahl der Parameter festgelegt. *Es müssen für das Format genügend Argumente vorhanden sein, da die Ergebnisse andernfalls unbestimmt und möglicherweise katastrophal sind.* Parameter, die durch den Formatstring nicht erfaßt (das heißt überzählig) sind, werden ignoriert.

Der Formatstring ist ein nullterminierter String, der zwei Arten von Informationen enthält:

- normale Zeichen, die direkt ausgegeben werden;
- Konvertierungsanweisungen, die die Auswertung korrespondierender Argumente und ihre Formatierung bestimmen.

Formatierungsanweisungen

Die Formatierungsanweisungen für *printf* haben die folgende Syntax:

```
% [Flags] [Breite] [.Präzision] [F|N|h|l|L] Typ
```

Jede Anweisung beginnt mit einem Prozentzeichen (%). Auf dieses Zeichen folgen (in der angegebenen Reihenfolge):

- eine (optionale) Zeichenfolge von Flag-Zeichen, `[Flags]`
- eine (optionale) Angabe über die minimal auszugebende Zeichenzahl, `[Breite]`
- eine (optionale) Angabe über die Genauigkeit, `[.Präzision]`
- eine (optionale) Angabe der Größe des Arguments, `[F|N|h|l|L]`
- ein Typzeichen, `[Typ]`

Optionale Bestandteile des Formatstrings

In der folgenden Tabelle sind die optionalen Bestandteile einer Formatierungsanweisung und ihre Bedeutung aufgelistet:

Zeichen	Wirkung
Flags	Ausrichtung der Ausgabe, Vorzeichen, Dezimalpunkt, folgende Nullen, oktale und hexadezimale Präfixe
Breite	Minimalzahl der ausgegebenen Zeichen, notfalls mit Leerzeichen oder Nullen aufgefüllt
Präzision	Maximalzahl der ausgegebenen Zeichen. Bei Integerwerten die Minimalanzahl der ausgegebenen Ziffern
Size	Verändern der Standardgröße des Arguments: **N = near**-Zeiger **F = far**-Zeiger **h = short int** **l = long** **L = long double**

Typzeichen

Die folgende Tabelle enthält die Typzeichen der *...printf*-Funktionen. Zusätzlich ist der Typ des jeweils akzeptierten Eingabearguments und das Format der Ausgabe angegeben.

Die Informationen in der Tabelle gelten nur, wenn keine Flags, keine Breiten- oder Präzisionsangabe und keine Größen-Modifizierer (s.o) angegeben wurden. Deren Auswirkung wird in den folgenden Tabellen berücksichtigt.

Typzeichen	Eingabeargurment	Ausgabeformat
Numerische Werte		
d	Integer	**signed int** (dezimal)
i	Integer	**signed int** (dezimal)
o	Integer	**unsigned int** (oktal)
u	Integer	**unsigned int** (dezimal)
x	Integer	**unsigned int** (hexadezimal), Buchstaben a bis f
X	Integer	**unsigned int** (hexadezimal), Buchstaben A bis F.
f	Gleitkomma	vorzeichenbehafteter Wert der Form [-]*dddd.dddd*
e	Gleitkomma	vorzeichenbehafteter Wert der Form [-]*d.dddd* **e**[+/-]*ddd*
g	Gleitkomma	vorzeichenbehafteter Wert im ***e***- oder ***f***-Format. *Präzision* gibt die Anzahl signifikanter Ziffern an. Abschließende Nullen und Dezimalpunkt werden nur ausgegeben, wenn es nötig ist.
E	Gleitkomma	dasselbe Format wie ***e***, aber mit dem Zeichen ***E*** vor dem Exponenten

G	Gleitkomma	dasselbe Format wie ***g***, aber mit dem Zeichen ***E*** vor dem Exponenten, wenn das Format ***e*** verwendet wird
Zeichen		
c	Zeichen	einzelnes Zeichen
s	String	Ausgabe bis zu einem Nullzeichen oder dem Erreichen der durch *Präzision* vorgegebenen Zeichenzahl
%	(nichts)	Ausgabe des Zeichens **%**.
Zeiger		
n	***int**	Speichert an der Position, auf die das Eingabeargument zeigt, die Anzahl der bis jetzt ausgegebenen Zeichen.
p	Zeiger	Gibt das Eingabeargument als Zeiger in hexadezimaler Form aus (**near**-Zeiger als YYYY, **far**-Zeiger als XXXX:YYYY).

Konventionen Einige dieser Formatierungsanweisungen sind von bestimmten Konventionen betroffen. Der bei der Ausgabe verwendete Dezimalpunkt wird von der aktuellen Locale-Kategorie LC_NUMERIC bestimmt. Die Konventionen werden in der folgenden Tabelle zusammengefaßt:

Zeichen	Konventionen
e oder **E**	Das Argument wird auf das Format [-]*d.ddd...***e**[+/-]*ddd* konvertiert, wobei ■ eine Ziffer vor dem Dezimalpunkt steht, ■ die Anzahl der Ziffern nach dem Dezimalpunkt mit der Präzisionsangabe übereinstimmt, ■ der Exponent zumindest aus zwei Ziffern besteht.
f	Das Argument wird in das dezimale Format *[-]ddd.ddd...* konvertiert. Die Anzahl der Ziffern nach dem Dezimalpunkt werden von der Präzisionsangabe bestimmt.
g oder **G**	Das Argument wird im Format **e**, **E** oder **f** ausgegeben. Die Präzisionsangabe legt die Anzahl der signifikanten Stellen fest. Abschließende Nullen werden vom Ergebnis entfernt und der Dezimalpunkt erscheint nur, wenn nötigt. Das Argument wird im Format **e** oder **f** ausgegeben, wenn **g** angegeben wurde, ansonsten wird das Format **E** verwendet. Das Format **e** wird nur dann verwendet, wenn der Exponent größer als die Präzisionsangabe oder kleiner als -4 ist.
x oder **X**	Beim Format **x** werden erscheinen die Hexadezimalziffern in kleinen, beim Format **X** in großen Buchstaben.

Unendlich große positive oder negative Gleitkommazahlen werden als +INF bzw. als -INF ausgegeben, eine »IEEE-Nichtzahl« als +NAN oder -NAN.

Flag-Zeichen *printf* kennt vier verschiedene Flag-Zeichen: Minus (-), Plus (+), Doppelkreuz (#) und Leerzeichen(). Flag-Zeichen können in beliebiger Folge und Kombination erscheinen und haben die folgenden Wirkungen:

Flag	Bedeutung
-	Linksbündige Ausgabe – nötigenfalls werden nach der Ausgabe Leerzeichen angefügt. Wenn dieses Flag nicht angegeben ist, arbeitet *printf* rechtsbündig, das heißt füllt links mit Leerzeichen oder Nullen auf.
+	Bei numerischer Ausgabe mit Vorzeichen wird das Vorzeichen stets mit ausgegeben. Ist dieses Flag nicht angegeben, stellt *printf* nur bei negativen Werten ein Minuszeichen voran.
Leerzeichen	Nichtnegative Werte beginnen mit einem Leerzeichen anstelle eines Pluszeichens. Negative Werte beginnen weiterhin mit einem Minuszeichen.
#	Konvertierung des Arguments in »alternativer Form« (siehe folgende Tabelle).

Wenn sowohl ein Leerzeichen als auch ein Pluszeichen angegeben sind, wird das Leerzeichen von *printf* ignoriert.

Alternative Formen

Die Verwendung des Flags # zusammen mit einem Typzeichen hat auf das zu konvertierende Argument (*arg*) folgende Wirkungen:

Typzeichen	Wirkung von # auf *arg*
c,s,d,i,u	# hat keine Wirkung.
o	Eine 0 wird vorangestellt, wenn *arg* ungleich 0 ist.
x,X	0x bzw. 0X wird vorangestellt.
e,E,f	Es wird immer ein Dezimalpunkt ausgegeben (normalerweise ist das bei diesen Formaten nur dann der Fall, wenn der Wert Stellen nach dem Komma enthält).
g,G	Wie bei ***e*** und ***E*** **–** zusätzlich werden abschließende Nullen nicht unterdrückt.

Breitenangabe

Diese Angabe legt die minimale Zahl auszugebender Zeichen fest.

Die Breitenangabe kann in zwei Formen erfolgen: entweder direkt innerhalb des Formatstrings oder indirekt über ein Sternchen (*). Bei der Verwendung von * erwartet *printf* ein zusätzliches Argument, das in der Liste direkt *vor* dem auszugebenden Wert stehen und den Typ **int** haben muß.

Wenn eine Ausgabe weniger Zeichen erzeugt als durch die Breitenangabe festgelegt ist, dann fügt *printf* (abhängig von den Flags) führende/abschließende Leerzeichen hinzu – erzeugt die Ausgabe dagegen mehr Zeichen als durch die Breitenangabe festgelegt, dann wird diese einfach ignoriert. Ausgaben werden durch die Breitenangabe in keinem Fall begrenzt oder abgeschnitten.

P

Breitenangabe	Wirkung
n	*printf* gibt mindestens *n* Zeichen aus und stellt nötigenfalls Leerzeichen voran (Standard) bzw. fügt abschließende Leerzeichen an (Flag -).
0n	*printf* gibt mindestens *n* Zeichen aus und stellt nötigenfalls eine entsprechende Anzahl führender Nullen voran.
*	Die Breitenangabe steht in der Parameterliste direkt vor dem auszugebenden Wert.

Präzisionsangabe Diese Angabe wird immer mit einem Dezimalpunkt (.) eingeleitet und so von einer eventuell vorangestellten Breitenangabe getrennt. Wie bei der Breitenangabe kann entweder eine direkte Angabe innerhalb des Formatstrings oder ein Sternchen (*) verwendet werden. Im letzteren Fall erwartet *printf* einen zusätzlichen Parameter des Typs **int**, der direkt *vor* dem auszugebenden Wert steht.

Wenn sowohl Breite als auch Präzision über Sternchen festgelegt werden, so erscheint in der Parameterliste zunächst die Breitenangabe, dann die Präzisionsangabe und unmittelbar danach das zugehörige Argument.

Präzisionsangabe	Wirkung auf die Ausgabe
keine	Die Standardwerte werden verwendet: 1 – für *d*, *i*, *o*, *u*, *x* und *X* 6 – für *e*, *E* und *f* alle signifikanten Stellen für *g* und *G* Ausgabe bis zum ersten Nullzeichen für *s* keine Auswirkungen für *c*
.0	Standardwerte für *d*, *i*, *o*, *u*, *x* und *X*, keine Ausgabe des Dezimalpunkts für *e*, *E* und *f*.
.n	Ausgabe von *n* Zeichen bzw. *n* signifikanten Dezimalstellen. Falls der Ausgabewert mehr als *n* Zeichen umfaßt, wird die Ausgabe unter Umständen abgeschnitten oder gerundet.
*	Die Präzisionsangabe ist in der Parameterliste enthalten und muß dort als **int** vor dem auszugebenden Wert stehen.

➥ Wenn die Präzision explizit mit 0 angegeben wird *und* es sich um ein Integerformat handelt (*d*, *i*, *o*, *u* oder *x*) *und* der auszugebende Wert 0 ist, dann wird für dieses Feld kein numerisches Zeichen ausgegeben (d.h. das Feld ist leer).

Typzeichen	Wirkung von .n auf die Konvertierung
d, *i*, *o*, *u*, *x*, *X*	Mindestens *n* Ziffern werden ausgegeben. Nötigenfalls stellt *printf* führende Nullen voran – falls der auszugebende Wert mehr Stellen umfaßt als durch *n* angegeben, wird der ausgegebene Wert nicht abgeschnitten.
e, *E*, *f*	*n* legt die Mindestanzahl der Stellen nach dem Komma fest. Die letzte Stelle wird gerundet.
g, *G*	*n* legt die Mindestanzahl signifikanter Dezimalstellen fest (inklusive der Stellen vor dem Komma).
c	Die Angabe von *n* hat keine Wirkung.
s	Es werden höchstens *n* Zeichen ausgegeben.

Größen-Modifizierer

Die Größenangaben (*F*, *N*, *h*, *l*, *L*) legen die Größe des folgenden Eingabearguments fest:

F = **far**-Zeiger
N = **near**-Zeiger
h = **short int**
l = **long**
L = **long double**

Die Größen-Modifizierer (*F*, *N*, *H*, *l*, *L*) legen fest, wie die *...printf* Funktionen den Datentyp des zugehörigen Arguments *arg* interpretieren. *F* und *N* werden nur bei Zeigerargumenten verwendet (%*p*, %*s* und %*n*). *h*, *l* und *L* werden nur bei numerischen Argumenten (Integer und Gleitkomma) verwendet.

Sowohl bei *F* als auch *N* wird das Eingabeargument *arg* neu interpretiert. Normalerweise ist *arg* bei den Typzeichen %*p*, %*s* oder %*n* ein Zeiger des aktuellen Speichermodells. *F* bedeutet nun "interpretiere *arg* als **far**-Zeiger" und *N* heißt "interpretiere *arg* als **near**-Zeiger".

h, *l* und *L* heben die Standardgröße von numerischen Eingabeargumenten auf: *l* und *L* beziehen sich auf Integer- (*d*, *i*, *o*, *u*, *x*, *X*) und Gleitkommatypen (*e*, *E*, *f*, *g*, *G*). Der Größen-Modifizierer *h* gilt dagegen aus- schließlich für Intergertypen. Weder *h* noch *l* haben Auswirkungen auf Zeichen (*c*, *s*) oder Zeigertypen (*p*, *n*)

Größen-Modifizierer	Wirkung auf die Interpretation von *arg*
F	*arg* wird als **far**-Zeiger gelesen.
N	*arg* wird als **near**-Zeiger gelesen.
h	*arg* wird als **short int** interpretiert (Formate *d*, *i*, *o*, *u*, *x*, *X*).
l	*arg* wird als **long int** (Formate *d*, *i*, *o*, *u*, *x*, *X*) bzw. als **double** interpretiert (Formate *e*, *E*, *f*, *g*, *G*).
L	Das Argument wird als **long double** interpretiert (Formate *e*, *E*, *f*, *g*, *G*).

Rückgabewert *printf* liefert bei fehlerfreier Ausführung die Anzahl der ausgegebenen Bytes, im Falle eines Fehlers EOF zurück.

Siehe auch *cprintf, ecvt, fprintf, fread, freopen, fscanf, putc, puts, putw, scanf, sprintf, vprintf, vsprintf*

putc — stdio.h

Funktion Schreibt ein Zeichen in einen Stream.

Syntax

```
int putc(int c, FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Das Makro *putc* schreibt das Zeichen *c* in den durch *stream* angegebenen Stream.

Rückgabewert *putc* liefert bei fehlerfreier Ausführung das übergebene Zeichen *c*, im Fehlerfall den Wert EOF zurück.

Siehe auch *fprintf, fputc, fputch, fputchar, fputs, fwrite, getc, getchar, printf, putch, putchar, putw, vprintf*

putch — conio.h

Funktion Gibt ein Zeichen auf dem Bildschirm aus.

Syntax

```
int putch(int c);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *putch* gibt das Zeichen *c* auf dem Bildschirm in das aktuelle Textfenster aus. Eine Übersetzung von Zeilenvorschubzeichen (\n) in CR/LF-Zeichenpaare findet nicht statt. *putch* funktioniert nur im Textmodus.

Abhängig vom Wert der globalen Variablen *_directvideo* wird der String entweder direkt in den Bildschirmspeicher geschrieben oder über den Umweg eines BIOS-Aufrufs ausgegeben.

➡ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert *putch* liefert bei fehlerfreier Ausführung das übergebene Zeichen *c*, im Fehlerfall den Wert EOF zurück.

Siehe auch *cprintf, cputs, getch, getche, putc, putchar*

putchar — stdio.h

Funktion *putchar* schreibt ein Zeichen in *stdout*.

Syntax

```
int putchar(int c);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *putchar* ist ein Makro und als *putc(c, stdout)* definiert.

Rückgabewert *putchar* liefert bei fehlerfreier Ausführung das übergebene Zeichen *c*, im Fehlerfall den Wert EOF zurück.

Siehe auch *fputchar, getc, getchar, printf, putc, putch, puts, putw, freopen, vprintf*

P

putenv — stdlib.h

Funktion Erweitert das aktuelle Environment um eine neue Umgebungsvariable.

Syntax

```
int putenv(const char *name);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *putenv* erwartet im String *name* die Definition einer Umgebungsvariablen (zum Beispiel `putenv("PATH=C:\\BC");` und fügt diese dem Environment des *aktuellen* Prozesses hinzu.

Mit *putenv* kann auch eine vorhandene Variable geändert oder gelöscht werden. Unter DOS und OS/2 muß *name* in Großbuchstaben angegeben werden. Unter anderen Systemen können Groß- und Kleinbuchstaben verwendet werden. Das Gleichheitszeichen (=) darf nicht in *name* enthalten sein. Eine Angabe der Form *name*= (das heißt ohne Zeichenfolge) löscht die Umgebungsvariable. Die von *putenv* erzeugte Umgebungsvariable können sowohl Groß- als auch Kleinbuchstaben enthalten.

putenv kann nur das Environment des laufenden Programmes verändern. Nach dem Programmende ist wieder das ursprüngliche Environment gültig. Das Environment des aktuellen Prozesses wird einschließlich aller mit *putenv* durchgeführten Änderungen an Kindprozesse übergeben.

Es dürfen nur statische oder globale Strings an *putenv* übergeben werden. Wird ein lokaler oder dynamischer String an *putenv* übergeben und benutzt, nachdem der Speicher des Strings freigegeben wurde, so kann dies zu unvorhersehbaren Ergebnissen führen.

Rückgabewert *putenv* liefert bei fehlerfreier Ausführung den Wert 0, andernfalls -1 zurück.

Siehe auch *getenv*

puts — stdio.h

Funktion Schreibt einen String in *stdout*.

Syntax

```
int puts(const char *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■		■

Beschreibung *puts* schreibt den nullterminierten String *s* zeichenweise in *stdout* und beendet die Ausgabe mit einem Zeilenvorschubzeichen.

➡ In Win32 oder Win32s GUI-Anwendungen muß *stdout* umgelenkt werden.

Rückgabewert Bei fehlerfreier Ausführung liefert die Funktion einen nicht-negativen Integerwert zurück. Im Fehlerfall ist der Rückgabewert EOF.

Siehe auch *cputs*, *fputs*, *gets*, *printf*, *putchar*, *freopen*

puttext — conio.h

Funktion Kopiert den Inhalt eines Speicherbereichs in einen rechteckigen Ausschnitt des Textbildschirms.

Syntax `int puttext(int left, int top, int right, int bottom, void *source);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

puttext schreibt den Inhalt des durch *source* bezeichneten Speicherbereichs in einen rechteckigen Ausschnitt des Textbildschirms. Der Ausschnitt wird durch die Parameter *left*, *top*, *right* und *bottom* festgelegt.

Die Funktion arbeitet mit absoluten Bildschirmkoordinaten. Die obere linke Ecke des Textbildschirms hat die Koordinaten (1,1).

puttext schreibt den Inhalt des angegebenen Speicherbereichs in sequentieller Reihenfolge in das definierte Rechteck von links nach rechts und von oben nach unten.

Jede Bildschirmposition benötigt zwei Speicher-Bytes. Das erste repräsentiert das dargestellte Zeichen und das zweite enthält das Video-Attribut dieser Bildschirmposition. Der für ein Rechteck mit *w* Spalten und *h* Zeilen benötigte Speicherplatz berechnet sich daher zu:

Bytes = (*h* Zeilen) x (*w* Spalten) x 2

puttext funktioniert nur im Textmodus und schreibt direkt in den Bildschirmspeicher.

➡ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Bei fehlerfreier Ausführung wird ein Wert ungleich Null zurückgeliefert. Ein Rückgabewert von 0 zeigt einen Fehler an (wie zum Beispiel Koordinaten außerhalb des Bildschirms).

Siehe auch *gettext*, *movetext*, *window*

P

putw — stdio.h

Funktion Schreibt einen Integer in einem Stream.

Syntax `int putw(int w, FILE *stream);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *putw* schreibt den Integer *w* in den Stream *stream*. Eine spezielle Anordnung in der Datei wird von der Funktion weder verursacht noch erwartet.

Rückgabewert *putw* liefert bei fehlerfreier Ausführung den übergebenen Integer *w*, im Fehlerfall den Wert EOF (-1) zurück. Da EOF einen gültigen Integerwert

darstellt, sollte nach *putw* ein expliziter Test mit *ferror* durchgeführt werden.

Siehe auch *getw, printf*

qsort — stdlib.h

Funktion Sortiert die Elemente eines Array mit Hilfe des Quicksort-Algorithmus.

Syntax

```
void qsort(void *base, size_t nelem, size_t width,
           int (_USERENTRY*fcmp)(const void *, const void *));
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *qsort* ist die Implementation der 3-Median-Variante des QuickSort-Algorithmus zum Sortieren der Elemente eines Array. *qsort* ruft dazu die vom Benutzer definierte Vergleichsfunktion *fcmp* wiederholt auf.

- *base* zeigt auf das erste Element (0) des zu sortierenden Array;
- *nelem* enthält die Anzahl der zu sortierenden Elemente;
- *width* gibt die Größe eines einzelnen Elements (in Bytes) an.

Die Vergleichsfunktion *fcmp* muß mit der _USERENTRY Aufrufskonvention verwendet werden.

fcmp übernimmt die beiden Argumente *elem1* und *elem2*. Beide sind Zeiger auf Elemente des Array. Die Vergleichsfunktion vergleicht die beiden Elemente (**elem1* und **elem2*) und liefert einen Integerwert als Ergebnis zurück:

- **elem1* < **elem2* — Rückgabewert < 0
- **elem1* == **elem2* — Rückgabewert = 0
- **elem1* > **elem2* — Rückgabewert > 0

Für den Vergleich bedeutet das Kleiner-als-Symbol (<), daß in der Sortierung das links stehende Element vor dem rechts stehenden erscheinen soll. Analog bedeutet das Größer-als-Symbol (>), daß in der Sortierung das links stehende Element nach dem rechts stehenden erscheinen soll.

Rückgabewert Keiner.

Siehe auch *bsearch, lsearch*

raise signal.h

Funktion Sendet ein Software-Signal zum ausführenden Programm.

Syntax

```
int raise(int sig);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Mit der Funktion *raise* lassen sich Software-Signale vom Typ *sig* an das Programm senden. Wenn das Programm einen Signal-Handler für den durch *sig* angegebenen Signaltyp installiert hat, wird diese Routine ausgeführt. Wenn kein Handler installiert ist, wird die Standardaktion für den jeweiligen Signaltyp ausgeführt.

Die gegenwärtig in signal.h definierten Signaltypen und Standardaktionen sind:

Signal	Bedeutung
SIGABRT	Nicht normales Programmende
SIGFPE	Gleitkommafehler
SIGILL	Illegaler Befehl
SIGINT	*Strg-C* Interrupt
SIGSEGV	Ungültiger Speicherzugriff
SIGTERM	Anfrage auf Programmbeendigung
SIGUSR1	Benutzerdefiniertes Signal
SIGUSR2	Benutzerdefiniertes Signal
SIGUSR3	Benutzerdefiniertes Signal
SIGBREAK	*Strg-Untbr* Interrupt

Der Signaltyp SIGABRT wird von Borland C++ normalerweise nicht erzeugt, kann jedoch durch *abort*, *raise* oder auch durch Exceptions ausgelöst werden, die nicht behandelt werden können.

P

Rückgabewert *raise* liefert bei fehlerfreier Ausführung den Wert 0, andernfalls einen Wert ungleich null zurück.

Siehe auch *abort*, *signal*

rand stdlib.h

Funktion Erzeugt eine Zufallszahl.

Syntax `int rand(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *rand* verwendet ein multiplikatives Kongruenzverfahren (Multiplikation mit anschließender Modulo-Operation), um Pseudozufallszahlen im Bereich von 0 bis RAND_MAX zu erzeugen. Die Periode des Zufallszahlengenerators beträgt 2^{32}. Die symbolische Konstante RAND_MAX ist in stdlib.h definiert.

Rückgabewert *rand* liefert die erzeugte Pseudozufallszahl zurück.

Siehe auch *random, randomize, srand*

random stdlib.h

Funktion Erzeugt eine Zufallszahl innerhalb eines bestimmten Bereichs.

Syntax `int random(int num);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *random* liefert eine Zufallszahl im Bereich von 0 bis (*num*-1) zurück und ist als Makro in stdlib.h definiert. Sowohl *num* als auch die zurückgelieferte Zufallszahl sind Integerwerte.

Rückgabewert *random* liefert einen Zahl zwischen 0 und (*num*-1) zurück.

Siehe auch *rand, randomize, srand*

randomize stdlib.h, time.h

Funktion Initialisiert den Zufallszahlengenerator.

Syntax `void randomize(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *randomize* initialisiert den Zufallszahlengenerator mit einem Zufallswert.

Rückgabewert Keiner.

Siehe auch *rand, random, srand*

_read io.h, dos.h

Veraltete Funktion. Siehe *_rtl_read*.

read io.h

Funktion Liest Daten aus einer Datei.

Syntax

```
int read(int handle, void *buf, unsigned len);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

read liest *len* Bytes aus der durch *handle* bezeichneten Datei in den durch *buf* angegebenen Speicherbereich.

Wird die Datei im Textmodus geöffnet, dann entfernt *read* Wagenrücklaufzeichen und interpretiert das Zeichen *Strg-Z* als Dateiende.

handle ist das über einen Aufruf von *creat*, *open*, *dup* oder *dup2* erhaltene Datei-Handle.

Bei Diskettendateien beginnt der Lesevorgang ab der aktuellen Position des Dateizeigers. Nach Beendigung des Lesevorgangs wird der Dateizeiger um die Anzahl der gelesenen Bytes erhöht. Bei Gerätedateien ist eine Position nicht definiert – hier liest *read* direkt vom entsprechenden Gerät.

read kann maximal UINT_MAX-1 Bytes lesen, da UINT_MAX selbst dem Wert -1 und damit dem Fehlerindikator entspricht. UINT_MAX ist in limits.h definiert.

Rückgabewert

Bei fehlerfreier Ausführung liefert *read* die Anzahl der gelesenen Bytes als positiven Integerwert zurück. Wenn die Datei im Textmodus geöffnet wurde, dann werden Wagenrückläufe und Strg-Z nicht mitgezählt.

Wenn das Dateiende erreicht bzw. überschritten wurde, so ist der Rückgabewert 0.

Im Fehlerfall ist der Rückgabewert -1 und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht gestattet
EBADF	Ungültiges Handle

Siehe auch *open*, *_rtl_read*, *write*

R

readdir dirent.h

Funktion Liest den aktuellen Eintrag aus einem Verzeichnis-Stream.

Syntax

```
struct dirent readdir(DIR *dirp);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *readdir* ist unter UNIX auf POSIX-Systemen verfügbar.

Die Funktion liest den aktuellen Verzeichniseintrag aus dem durch *dirp* bezeichneten Verzeichnis-Stream. Der Verzeichnis-Stream wird auf den nächsten Eintrag gesetzt.

readdir liefert einen Zeiger auf eine Struktur des Typs *dirent* zurück. Bei jedem neuen Aufruf von *readdir* für den gleichen Verzeichnis-Stream wird diese Struktur überschrieben.

Die in dirent.h definierte Struktur des Typs *dirent* entspricht einem einzelnen Verzeichniseintrag. Sie enthält mehrere Elemente, von denen der Programmierer aber nur auf eines zugreifen kann:

```
char d_name[];
```

Das **char**-Array *d_name* enthält den nullterminierten Dateinamen für den aktuellen Verzeichniseintrag. Die Größe dieses Array ist unbestimmt. Sie können die Länge des Dateinamens mit der Funktion *strlen* ermitteln.

Zurückgeliefert werden alle Verzeichniseinträge einschließlich Unterverzeichnissen, ».«- und »..«-Einträgen, Systemdateien, verborgenen Dateien und Datenträgernamen. Unbenutzte oder gelöschte Verzeichniseinträge werden nicht zurückgeliefert.

Verzeichniseinträge können zwar während des Lesens eines Verzeichnis-Stream erstellt oder gelöscht werden, es ist jedoch nicht sicher, ob *readdir* die betreffenden Einträge zurückliefert. Um sicherzustellen, daß *readdir* den aktuellen Status des Verzeichnisses zurückliefert, muß der Verzeichnis-Stream mit *rewinddir* auf den ersten Eintrag zurückgesetzt oder das Verzeichnis mit *opendir* neu geöffnet werden.

Rückgabewert Bei fehlerfreier Ausführung liefert *readdir* einen Zeiger auf den aktuellen Verzeichniseintrag zurück. *readdir* liefert NULL zurück, wenn das Verzeichnisende erreicht wird oder *dirp* auf keinen offenen Verzeichnis-Stream zeigt.

Siehe auch *closedir*, *opendir*, *rewinddir*

realloc stdlib.h

Funktion Ändert die Größe eines reservierten Blocks im Hauptspeicher.

Syntax

```
void *realloc(void *block, size_t size);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *realloc* verändert die Größe eines zuvor reservierten Speicherblocks auf *size* Bytes. Enthält *size* den Wert null, so wird der Speicherblock freigegeben und NULL zurückgeliefert. *block* muß auf die Startadresse eines Speicherblocks zeigen, der zuvor mit einem Aufruf von *malloc*, *calloc* oder *realloc* ermittelt wurde. Wenn *block* ein NULL-Zeiger ist, verhält sich *realloc* wie *malloc*.

realloc ändert die Größe des reservierten Speicherblocks und kopiert dazu den Blockinhalt an eine neue Adresse, falls nötig.

Rückgabewert *realloc* gibt die neue Adresse des Speicherblocks zurück, die von der bisherigen Adresse abweichen kann. Ist die Größenänderung nicht möglich, gibt *realloc* NULL zurück.

Siehe auch *calloc*, *farrealloc*, *free*, *malloc*

remove stdio.h

Funktion Löscht eine Datei.

Syntax

```
int remove(const char *filename);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *remove* löscht die durch *filename* angegebene Datei. Das Makro *remove* wird dabei in einen Aufruf von *unlink* umgesetzt. Wenn die Datei offen ist, müssen Sie sie erst schließen, bevor Sie sie mit *remove* löschen können.

Der Dateiname kann einen vollständigen Pfad enthalten.

Rückgabewert Bei fehlerfreier Ausführung liefert *remove* den Wert 0 zurück. Im Fehlerfall ist der Rückgabewert -1 und die globale Variable *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
ENOENT	Pfad/Datei nicht gefunden

Siehe auch *unlink*

rename stdio.h

Funktion Ändert den Namen einer Datei.

Syntax

```
int rename(const char *oldname, const char *newname);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *rename* ändert den Namen der Datei *oldname* in *newname*. Beide Namen können vollständige Pfade und/oder Laufwerksbezeichner enthalten. Die Angabe unterschiedlicher Laufwerke in *oldname* und *newname* ist nicht zulässig.

Es ist mit *rename* möglich, Dateien von einem Verzeichnis in ein anderes zu verlagern. Die Angabe von Jokerzeichen ist nicht zulässig.

Ist eine der Dateien in irgendeinem Prozeß geöffnet, so kehrt *rename* mit einem Fehler (EACESS) zurück.

Rückgabewert Bei fehlerfreier Ausführung liefert *rename* den Wert 0 zurück; ansonsten ist das Funktionsergebnis -1 und *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
ENOENT	Pfad/Datei nicht gefunden
ENOTSAM	Ziel- und Quell-Laufwerk sind nicht identisch

rewind stdio.h

Funktion Setzt einen Dateizeiger auf den Stream-Anfang.

Syntax

```
void rewind(FILE *stream);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *rewind* ist äquivalent zu *fseek*(*stream*, 0L, SEEK_SET). Anders als *fseek* löscht *rewind* nicht nur das Dateiende-Flag, sondern auch eventuell gesetzte Fehler-Flags.

In Dateien, die für Lese- *und* Schreiboperationen geöffnet worden sind, kann nach *rewind* zwischen »Lesen« und »Schreiben« gewechselt werden.

Rückgabewert Keiner.

Siehe auch *fopen*, *fseek*, *ftell*

rewinddir dirent.h

Funktion Setzt einen Verzeichnis-Stream auf den ersten Eintrag zurück.

Syntax

```
void rewinddir(DIR *dirp);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *rewinddir* ist unter UNIX auf POSIX-Systemen verfügbar.

Die Funktion setzt den Verzeichnis-Stream *dirp* auf den ersten Eintrag im Verzeichnis zurück. *rewinddir* stellt sicher, daß der Verzeichnis-Stream den aktuellen Status der Einträge widerspiegelt: Es werden alle Einträge berücksichtigt, die seit dem letzten Aufruf von *opendir* oder *rewinddir* für diesen Verzeichnis-Stream erzeugt oder gelöscht wurden.

Rückgabewert Keiner.

Siehe auch *closedir*, *opendir*, *readdir*

rmdir dir.h

Funktion Löscht ein Verzeichnis.

Syntax

```
int rmdir(const char *path);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *rmdir* löscht das durch *path* angegebene Verzeichnis und kann nur angewendet werden, wenn dieses Verzeichnis

- keine Einträge enthält,
- nicht das aktuelle Arbeitsverzeichnis ist und
- nicht das Stammverzeichnis der Diskette/Festplatte darstellt.

Rückgabewert *rmdir* liefert 0 zurück, wenn das Verzeichnis erfolgreich gelöscht wurde. Im Fehlerfall ist der Rückgabewert -1 und *errno* bekommt einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
ENOENT	Pfad nicht gefunden

Siehe auch *chdir*, *getcurdir*, *getcwd*, *mkdir*

R

rmtmp — stdio.h

Funktion Löscht temporäre Dateien.

Syntax

```
int rmtmp(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *rmtmp* schließt und löscht alle offenen, temporären Datei-Streams, die vorher mit *tmpfile* erzeugt wurden. Das aktuelle Verzeichnis muß dasselbe sein, in dem die Dateien erzeugt wurden, da die Dateien andernfalls nicht gelöscht werden.

Rückgabewert *rmtmp* liefert die Anzahl der geschlossenen und gelöschten Dateien zurück.

Siehe auch *tmpfile*

_rotl, _rotr — stdlib.h

Funktion Rotiert einen **unsigned short**-Wert bitweise nach links oder rechts.

Syntax

```
unsigned _rotl(unsigned value, int count);
unsigned _rotr(unsigned value, int count);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_rotl* rotiert den als *value* übergebenen Integerwert um *count* Bitpositionen nach links.

_rotr rotiert den als *value* übergebenen Integerwert um *count* Bitpositionen nach rechts.

Rückgabewert Die Funktion liefert den rotierten Integerwert zurück:

- *_rotl* liefert den um *count* Bits nach links rotierten Wert von *value* zurück.
- *_rotr* liefert den um *count* Bits nach rechts rotierten Wert von *value* zurück.

Siehe auch *_crotl*, *_crotr*, *_lrotl*, *_lrotr*

_rtl_chmod — dos.h, io.h

Funktion Liest bzw. ändert die DOS-Dateiattribute.

Syntax

```
int _rtl_chmod(const char *path, int func[, int attrib]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung

Über *_rtl_chmod* kann die Zugriffsberechtigung der durch *path* angegebenen Datei entweder gelesen (*func* == 0) oder auf den Wert von *attrib* gesetzt werden (*func* == 1). Für den Parameter *attrib* sind die folgenden Konstanten in dos.h definiert:

FA_RDONLY	Nur Lesen
FA_HIDDEN	Verborgen
FA_SYSTEM	Systemdatei
FA_LABEL	Volume-Label (Datenträgername)
FA_DIREC	Verzeichnis
FA_ARCH	Archiv

Rückgabewert

Bei fehlerfreier Ausführung liefert *_rtl_chmod* das gelesene bzw. neu gesetzte Attribut.

Im Fehlerfall ist der Rückgabewert -1 und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht erlaubt
ENOENT	Pfad/Datei nicht gefunden

Siehe auch

chmod, *_rtl_creat*

_rtl_close, close — io.h

Funktion

Schließt eine Datei.

Syntax

```
int _rtl_close(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung

_rtl_close schließt die durch *handle* bezeichnete Datei. Ein Handle wird durch einen Aufruf von *_rtl_creat*, *creat*, *creatnew*, *creattemp*, *dup*, *dup2*, *open* oder *_rtl_open* einer Datei zugeordnet.

Diese Funktion schreibt kein ^Z (*Ctrl-Z*) als Dateiende-Zeichen. Falls Sie eine Datei mit ^Z abschließen wollen, müssen Sie dieses Zeichen explizit ausgeben.

Rückgabewert

_rtl_close liefert den Wert 0 zurück, wenn die Operation fehlerfrei ausgeführt werden konnte, ansonsten -1.

handle muß ein gültiges Handle sein, dem eine gültige, offene Datei zugeordnet ist. Ist dies nicht der Fall, erhält die globale Variable *errno* den Wert:

EBADF	Ungültiges Handle

Siehe auch *chzize, close, creatnew, dup, fclose, open, _rtl_creat, _rtl_open, sopen*

_rtl_creat dos.h, io.h

Funktion Erzeugt eine neue Datei oder überschreibt eine bereits vorhandene Datei.

Syntax

```
int _rtl_creat(const char *path, int attrib);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung Die Funktion *_rtl_creat* öffnet die durch *path* bezeichnete Datei grundsätzlich im binären Modus. Nach einer fehlerfreien Öffnung bzw. Erzeugung zeigt der Dateizeiger auf das erste Byte der Datei. Die Datei wird für Lese- und Schreibzugriffe geöffnet.

Wenn die Datei bereits existiert, wird sie auf null Bytes Länge gekürzt (d.h. der vorherige Inhalt wird gelöscht).

Der Parameter *attrib* enthält eine ODER-Kombination aus einer oder mehreren der folgenden in dos.h definierten Konstanten:

FA_RDONLY	Nur-Lesen-Attribut
FA_HIDDEN	Verborgen
FA_SYSTEM	Systemdatei

Rückgabewert Bei fehlerfreier Ausführung liefert *_rtl_creat* das neue Datei-Handle (einen positiven Integerwert) zurück. Ansonsten ist der Rückgabewert -1.

Im Fehlerfall wird die globale Variable *errno* auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht erlaubt
ENOENT	Pfad/Datei nicht gefunden
EMFILE	Zu viele offene Dateien

Siehe auch *chsize, close, creat, creatnew, creattemp, _rtl_chmod, _rtl_close*

_rtl_heapwalk malloc.h

Funktion Durchläuft den Heap Knoten für Knoten.

Syntax

```
int _rtl_heapwalk(_HEAPINFO *hi);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			

Beschreibung

_rtl_heapwalk durchläuft den Heap Knoten für Knoten. Die Funktion geht davon aus, daß der Heap gültig ist (überprüfen Sie dies vorher mit *_heapchk*). Mit dem letzten Block wird _HEAPOK zurückgeliefert. Bei einem erneuten Aufruf von *_rtl_heapwalk* ist das Ergebnis _HEAPEND.

Die Funktion *_rtl_heapwalk* übernimmt einen Zeiger auf eine Struktur vom Typ *_HEAPINFO* (deklariert in malloc.h). Setzen Sie das Feld *hi._pentry* beim ersten Aufruf der Funktion auf NULL. Nach der Rückkehr enthält *hi._pentry* die Adresse des ersten Blocks, während *hi._size* die Größe des Blocks in Bytes angibt.

Wird der Block zur Zeit benutzt, ist das Flag *hi._useflag* auf den Wert _USEDENTRY gesetzt. Bei einem freien Block enthält *hi._useflag* den Wert _FREEENTRY.

Rückgabewert

Einer der folgenden Werte wird zurückgeliefert:

_HEAPBADNODE	Ein fehlerhafter Block wurde entdeckt
_HEAPBADPTR	Das Feld *_pentry* zeigt nicht auf einen gültigen Heap-Block
_HEAPEMPTY	Es gibt keinen Heap
_HEAPEND	Das Ende des Heap wurde erreicht
_HEAPOK	Der *_HEAPINFO*-Block enthält gültige Daten über den nächsten Block

Siehe auch

_heapchk, *_heapset*

_rtl_open — fcntl.h, share.h, dos.h

Öffnet eine Datei für Lese- oder Schreibaktionen.

Syntax

```
int _rtl_open(const char *filename, int oflags);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung

_rtl_open öffnet die durch *filename* angegebene Datei entsprechend des Parameters *oflags* für Lese- und/oder Schreibzugriffe. Die Datei wird stets im Binärmodus geöffnet.

oflags verwendet die Flags der folgenden beiden Listen. Aus der ersten Liste darf (und muß) genau ein Flag verwendet werden. Die übrigen Flags können in jeder logischen Kombination benutzt werden.

Liste 1: Lese-/Schreib-Flags

O_RDONLY	Nur Lesen
O_WRONLY	Nur Schreiben
O_RDWR	Lesen und Schreiben

Die folgenden zusätzlichen Werte können druch eine ODER-Operation mit in *oflags* aufgenommen werden:

Diese symbolischen Konstanten sind in fcntl.h und in share.h definiert.

Liste 2: Weitere Zugriffs-Flags

O_NOINHERIT	Datei wird nicht an Kindprozesse vererbt
SH_COMPAT	Ermöglicht weitere Zugriffe auf diese Datei mit SH_COMPAT. Wurde die Datei bereits in einem gemeinsamen Zugriffsmodus geöffnet, ergibt der Aufruf ein Fehler.
SH_DENYRW	Ermöglicht den Zugriff auf die Datei nur über das aktuelle Handle.
SH_DENYWR	Über andere Handles kann diese Datei nur gelesen, aber nicht beschrieben werden.
SH_DENYRD	Über andere Handles kann diese Datei nur beschrieben werden.
SH_DENYNONE	Sämtliche Zugriffe auf diese Datei sind erlaubt, jedoch nicht solche mit SH_COMPAT.

In einem Aufruf von *_rtl_open* kann jeweils nur eine der SH_DENYxxx-Konstanten angegeben werden. Diese Konstanten gelten zusätzlich zu bereits bestehenden Sperrungen der Datei.

Die maximal mögliche Anzahl offener Dateien wird durch HANDLE_MAX definiert.

Rückgabewert

Wenn die Datei fehlerfrei geöffnet werden konnte, liefert *_rtl_open* das Handle der Datei zurück (einen **int**-Wert größer 0). Der Dateizeiger – er markiert die aktuelle Position in der Datei – wird auf den Dateianfang gesetzt.

Im Fehlerfall liefert *_rtl_open* den Wert -1 zurück und setzt die globale Variable *errno* auf einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
EINVACC	Ungültiger Zugriffscode
ENOENT	Pfad oder Datei nicht gefunden
EMFILE	Zu viele Dateien offen

Siehe auch

open, *_rtl_read*, *sopen*

_rtl_read io.h, dos.h

Liest Daten aus einer Datei.

Syntax

```
int _rtl_read(int handle, void *buf, unsigned
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

_rtl_read liest *len* Bytes aus der durch *handle* bezeichneten Datei in den durch *buf* angegebenen Speicherbereich.

Wurde die Datei im Textmodus geöffnet, dann entfernt *_rtl_read* keine CR-Zeichen.

handle ist das über einen Aufruf von *creat*, *open*, *dup* oder *dup2* erhaltene Datei-Handle.

Bei Diskettendateien beginnt der Lesevorgang ab der aktuellen Position des Dateizeigers. Nach Beendigung des Lesevorgangs wird der Dateizeiger um die Anzahl der gelesenen Bytes erhöht. Bei Gerätedateien ist eine Position nicht definiert – hier liest *_rtl_read* direkt vom entsprechenden Gerät.

_rtl_read kann maximal UINT_MAX-1 Bytes lesen, da UINT_MAX selbst dem Wert -1 und damit dem Fehlerindikator entspricht. Die Konstante UINT_MAX ist in limits.h definiert.

Rückgabewert

_rtl_read liefert bei fehlerfreier Ausführung die Anzahl der gelesenen Bytes als (positiven) Integerwert zurück. Wenn das Dateiende erreicht bzw. überschritten wurde, liefert *_rtl_read* den Wert 0 zurück. Im Falle eines Fehlers liefert *_rtl_read* den Wert -1 zurück und die globale Variable *errno* wird auf einen der folgenden Werte gesetzt:

EACCES	Zugriff nicht gestattet
EBADF	Ungültiges Handle

Siehe auch

read, *_rtl_open*, *_rtl_write*

R

_rtl_write io.h

Funktion

Schreibt in eine Datei.

Syntax

```
int _rtl_write(int handle, void *buf, unsigned len);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung

_rtl_write liest *len* Bytes ab der durch *buf* angegebenen Adresse aus dem Speicher und schreibt sie in die durch *handle* bezeichnete Datei. Mit *_rtl_write* können maximal UINT_MAX-1 Bytes geschrieben werden, da UNIT_MAX dem Wert -1 entspricht, dem Fehlerindikator von *write*. UINT_MAX ist in limits.h definiert. Wenn die Datei im Textmodus geöffnet wurde und *write* auf das Zeilenvorschubzeichen LF ('\n') trifft, so wird diese Zeichenfolge *nicht* in CR/LF umgesetzt.

Ist die Anzahl der tatsächlich geschriebenen Bytes kleiner als *len*, so ist dies ein Fehler und bedeutet wahrscheinlich, daß die Diskette/Festplatte voll ist.

Bei Diskettendateien schreibt *_rtl_write* ab der aktuellen Position des Dateizeigers, bei Gerätedateien wird direkt auf das jeweilige Gerät geschrieben.

Wenn die Datei mit O_APPEND (anhängen) geöffnet wurde, setzt *_rtl_write* den Dateizeigers automatisch auf das Dateiende.

Rückgabewert

Die Funktion liefert die Anzahl der tatsächlich geschriebenen Bytes zurück. Im Fehlerfall wird der Wert -1 zurückgeliefert und *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
EBADF	Ungültiges Handle

Siehe auch

lseek, *_rtl_read*, *write*

scanf — stdio.h

Funktion

Liest und formatiert Eingabe aus dem Stream *stdin*.

Syntax

```
int scanf(const char *format[, address, ...]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■	■	■	■

Beschreibung

scanf liest eine Reihe von Eingabefeldern zeichenweise aus dem Stream *stdin*. Jedes Feld wird dann anhand von Formatierungsanweisungen formatiert, die *scanf* über den Zeiger *format* als Formatstring übergeben wurden. Die dem Parameter *format* folgenden Parameter bezeichnen Adressen, an denen die gelesenen und formatierten Eingaben gespeichert werden. Die Anzahl der Eingabefeldern muß mit der Anzahl der Formatierungsanweisungen und Adressen übereinstimmen.

➡ In Win32 oder Win32s GUI-Anwendungen muß *stdin* umgelenkt werden.

Der Formatstring

Der Formatstring ist in jeder Variante von *scanf* (*cscanf*, *fscanf*, *sscanf*, *vscanf*, *vfscanf*, *vsscanf*) obligatorisch – über ihn wird die Art der Konvertierung, die Formatierung und die Anzahl der Argumente festgelegt. *Es müssen mindestens so viele Adreßparameter wie Formatierungsanweisungen vorhanden sein, da die Ergebnisse ansonsten unbestimmt und möglicherweise katastrophal sind*. Über- zählige Adreßparameter (mehr als durch das Format angegeben) werden ignoriert.

scanf führt oft zu überraschenden Ergebnissen, wenn die Eingabe vom erwarteten Muster abweicht. Sie müssen genau angeben, wie *scanf* das Zeilenende behandeln soll. Die Kombination von *gets* oder *fgets* mit einem folgenden *sscanf* ist sicher und einfach und deshalb vorzuziehen.

Der Formatstring ist ein nullterminierter String, der drei Arten von Informationen enthält:

- Whitespace-Zeichen (Leerzeichen, Tabulatoren (\t) und Zeilenvorschübe (\n)) weisen *scanf* an, dieses und alle folgenden Whitespace-Zeichen zwar zu lesen, aber nicht zu speichern;
- normale Zeichen, d.h. alle anderen ASCII-Zeichen mit Ausnahme des Prozentzeichens (%), die im Prinzip dieselbe Wirkung wie die Whitespace-Zeichen haben. Findet eine *...scanf*-Funktion ein normales Zeichen im Formatstring, so wird ein entsprechendes normales Zeichen in der Eingabe zwar gelesen aber nicht gespeichert.
- Formatierungsanweisungen, die *scanf* mitteilen, welche Art von Eingabe erwartet wird und wie dieses Eingabe konvertiert und an den durch die Adreßparameter bestimmten Adressen gespeichert werden soll.

Nachfolgende Whitespace-Zeichen (einschließlich Zeilenvorschüben) werden nur gelesen, wenn dies im Formatstring angegeben wird.

Formatierungsanweisungen

Die Formatierungsanweisungen für *printf* haben die folgende Syntax:

```
% [*] [Breite] [F|N] [h|l|L] Typ
```

Jede Anweisung beginnt mit einem Prozentzeichen (%). Auf dieses Zeichen folgen (in der angegebenen Reihenfolge):

R

- ein optionales Zeichen, mit dem die Zuweisung des nächsten Eingabefelds unterdrückt wird, `[*]`
- eine optionale Breitenangabe, `[Breite]`
- eine optionale Angabe der Zeigergröße, `[F|N]`
- eine optionale zusätzliche Typangabe, `[h|l|L]`
- das Typzeichen

Optionale Bestandteile des Formatstrings

In der folgenden Tabelle werden die Auswirkungen der optionalen Angaben im *...scanf*-Formatstring erläutert:

Zeichen oder Angabe	Wirkung oder Bedeutung
*	Die Zuweisung des nächsten Eingabefeldes wird unterdrückt.
Breite	Maximale Anzahl der zu lesenden Zeichen. Wenn vor Erreichen dieser Zeichenzahl ein Whitespace-Zeichen oder ein anderes nicht auswertbares Zeichen gelesen wird, beendet *scanf* die Eingabe dieses Wertes von selbst.
F\|N	Angabe der Größe des Adreßparameters: *N* = **near**-Zeiger *F* = **far**-Zeiger
h\|l\|L	Änderung des Typs des Adreßparameters: *h* = **short int** *l* = **long int**, falls das Typzeichen eine Integer-Konvertierung angibt *l* = **double**, falls das Typzeichen eine Gleitkomma-Konvertierung angibt *L* = **long double** (nur bei Gleitkommakonvertierungen)

Typzeichen von ...scanf

In der folgenden Tabelle werden die Typzeichen der *...scanf*-Funktionen zusammen mit dem jeweils erwarteten Eingabetyp und dem gespeicherten Format aufgelistet.

Die folgende Aufstellung geht davon aus, daß in der Formatierungsanweisung außer dem Typ keine weiteren, optionalen Angaben (*, Breite oder Zeigergröße) existieren.

Die zusätzlichen Auswirkungen dieser optionalen Elemente können Sie den nachfolgenden Tabellen entnehmen.

Typ	erwartete Eingabe	Adreßparameter
Numerische Werte		
d	Integer (dezimal)	Zeiger auf **int** (**int** **arg*)
D	Integer (dezimal)	Zeiger auf **long** (**long** **arg*)
o	Integer (oktal)	Zeiger auf **int** (**int** **arg*)
O	Integer (oktal)	Zeiger auf **long** (**long** **arg*)
i	Integer (dezimal, oktal oder hexadezimal)	Zeiger auf **int** (**int** **arg*)
I	Integer (dezimal, oktal oder hexadezimal)	Zeiger auf **long** (**long** **arg*)
u	Integer (dezimal)	Zeiger auf **unsigned int** (**unsigned int** **arg*)
U	Integer (dezimal)	Zeiger auf **unsigned long**, (**unsigned long** **arg*)
x	Integer (hexadezimal)	Zeiger auf **unsigned int**, (**unsigned int** **arg*)
x	Integer (hexadezimal)	Zeiger auf**unsigned long**, (**unsigned long** **arg*)
e,E	Gleitkomma	Zeiger auf **float** (**float** **arg*)
f	Gleitkomma	Zeiger auf **float** (**float** **arg*)
g,G	Gleitkomma	Zeiger auf **float** (**float** **arg*)

Zeichen		
s	String	Zeiger auf einen String (**char** *arg*[])
c	Zeichen	Zeiger auf Zeichen (**char** **arg*). Wird zusammen mit dem Typzeichen *c* eine Breite *B* angegeben (beispielsweise %5c): Zeiger auf ein **char**-Array der Größe *B* (**char** *arg*[B]).
%	Zeichen %	keine Konvertierung, sondern Speicherung des Zeichens %
Zeiger		
n		Zeiger auf **int**. Im entsprechenden Parameter (**int** **arg*) wird die Anzahl der bis jetzt gelesenen Zeichen gespeichert.
p	YYYY oder YYYY:ZZZZ (hexadezimal)	Zeiger auf ein beliebiges Objekt (**far*** oder **near***). Die Konvertierung richtet sich nach dem verwendeten Speichermodell

Eingabefelder Ein Eingabefeld besteht aus:

- allen Zeichen bis zum nächsten Whitespace-Zeichen, wobei das Whitespace-Zeichen nicht mit eingeschlossen ist;
- oder allen Zeichen bis zum ersten Zeichen, das sich über die aktuelle Formatierungsanweisung nicht auswerten läßt (wie zum Beispiel die Zeichen »8« und »9« bei der Eingabe eines oktalen Wertes);
- oder maximal so vielen Zeichen, wie in der aktuellen Formatierungsanweisung als Breite angegeben ist.

Konventionen Einige dieser Formatierungsanweisungen sind von bestimmten Konventionen betroffen. Der bei der Ausgabe verwendete Dezimalpunkt wird von der aktuellen Locale-Kategorie LC_NUMERIC bestimmt. Die Konventionen werden im folgenden zusammengefaßt:

%c-Format

Die Anweisung *%c* liest das jeweils nächste Zeichen, wobei Whitespace-Zeichen nicht gesondert berücksichtigt (oder überlesen) werden. Um Whitespace-Zeichen zu überspringen und das folgende Zeichen zu lesen, sollte man deshalb die Anweisung *%1s* verwenden.

%Bc-Format (B = Breite)

scanf erwartet einen Zeiger auf ein **char**-Array mit *B* Elementen als Adreßargument (char *arg*[B]).

%s-Format

scanf erwartet einen Zeiger auf ein **char**-Array als Adreßargument (char *arg*[]).

Das Array muß Platz für ein zusätzliches Nullzeichen enthalten. Die Eingabe wird durch ein Leerzeichen oder einen Zeilenvorschub beendet, das selbst nicht gespeichert wird. *scanf* speichert automatisch ein Nullzeichen als letztes Zeichen des Strings.

R

%[Suchmenge]-Format

scanf erwartet (wie bei %s) einen Zeiger auf ein **char**-Array als Adreßparameter.

Suchmenge ist eine Folge von Zeichen, aus denen das Eingabefeld bestehen muß.

Ist das erste Zeichen der Suchmenge ein Caret-Zeichen (^), dann wird die Suchmenge zu der Menge aller Zeichen mit Ausnahme der angegegeben Zeichen invertiert. (Das Caret-Zeichen selbst ist dann normalerweise Bestandteil der Suchmenge, wenn es nicht explizit ein weiteres Mal nach dem ersten Caret aufgeführt wird.)

Das Eingabefeld wird in diesem Fall nicht durch Whitespace-Zeichen begrenzt. *...scanf* liest alle Zeichen bis zum ersten Zeichen, das nicht in der Suchmenge (bzw. invertierten Suchmenge) enthalten ist. Zwei Beispiele:

`%[abcd]`	sucht im Eingabefeld nach den Zeichen *a*, *b*, *c* und *d*;
`%[^abcd]`	sucht im Eingabefeld nach allen Zeichen außer *a*, *b* , *c* oder *d*.

Anstelle einer direkten Aufzählung der Suchmenge kann man als Kurzschreibweise auch einen Zeichenbereich angeben. Um beispielsweise alle Dezimalziffern zu erfassen, können Sie die Suchmenge mit `%[0123456789]` angeben oder Sie können die äquivalenten Kurzschreibweise `%[0-9]` verwenden.

Soll die Suchmenge alle alphanumerischen Zeichen enthalten, so benutzen Sie folgende Kurzschreibweisen:

`%[A-Z]`	sämtliche Großbuchstaben
`%[0-9A-Za-z]`	sämtliche Groß- und Kleinbuchstaben sowie Ziffern
`%[A-Za-zäöüÄÖÜß]`	sämtliche Buchstaben inklusive der Umlaute und des ß.

Die Regeln zur Angabe von Bereichen sind recht einfach:

- Das Zeichen vor dem Bindestrich (-) muß einen ASCII-Code haben, der mindestens um eins niedriger ist als der des auf den Bindestrich folgenden Zeichens.
- Der Bindestrich darf weder das erste noch das letzte Zeichen der Aufzählung sein (andernfalls wird er nicht als Bereichsangabe, sondern als Bindestrich gewertet).
- Das Zeichen vor und das Zeichen nach dem Bindestrich muß jeweils den Beginn bzw. das Ende eines Bereichs darstellen, dürfen also nicht gleichzeitig Beginn und Ende zweier verschiedener Bereiche sein.

In allen anderen Fällen wird der Bindestrich selbst als Zeichen (und nicht als Bereichsangabe) gewertet. Einige Beispiele dazu:

`%[-+*/]`	die vier arithmetischen Operatoren
`%[z-a]`	die Zeichen »z«, »-« und »a«
`%[+0-9-A-F]`	das Zeichen »+«, die Ziffern von 0 bis 9, das Zeichen »-« und die Buchstaben von A bis F
`%[+0-9A-F-]`	dieselben Zeichen wie im vorigen Beispiel
`%[^-0-9+A-F]`	sämtliche Zeichen außer »-«, den Ziffern 0 bis 9, dem Zeichen »+« und den Buchstaben A bis F.

Formate %e, %E , %f , %g und %G (Gleitkomma)
Gleitkommazahlen müssen dem folgenden Format entsprechen:

```
[+|-] dddddddd [.] dddd [E|e] [+|-] ddd
```

d steht hier für dezimale, oktale oder hexadezimale Ziffern, in eckige Klammern eingefaßte Teile sind optional.

INF = INFinity (unendlich); NAN = Not-A-Number (keine Zahl)

Zusätzlich werden **+INF**, **-INF**, **+NAN** und **-NAN** als Gleitkommazahlen erkannt. Hierfür sind sowohl Vorzeichen und Großschreibung erforderlich.

Formate %d, %i, %o, %x, %D, %I, %O, %X, %c und %n
In allen Fällen, in denen Zeiger auf **char**, **int** oder **long** erlaubt sind, sind auch Zeiger auf die jeweiligen **unsigned**-Typen erlaubt.

Unterdrückung einer Zuordnung mit *

Folgt das Zeichen * direkt auf das Zeichen % einer Formatierungsanweisung, so wertet *scanf* die restlichen Elemente dieser Anweisung aus und analysiert die Eingabe, weist diese jedoch nicht dem nächsten Adreßparameter zu. (Hier wird zwar dasselbe Zeichen wie für den Umleitungsoperator verwendet – ansonsten haben diese beiden Dinge aber nichts miteinander zu tun).

Die Unterdrückung von Zuweisungen bei *scanf* ist mit gewisser Vorsicht zu genießen, da es keine direkte Möglichkeit gibt festzustellen, ob die »richtigen« Teile einer Eingabe dadurch übersprungen wurden.

Beitenangaben

Die Breitenangabe erfolgt durch einen dezimalen Integerwert innerhalb der Formatierungsanweisung, der zwischen dem % und dem Typ der zu erwartenden Eingabe steht (*%18s* legt beispielsweise fest, daß *scanf* einen String mit maximal 18 Zeichen einlesen soll).

Enthält das Eingabefeld weniger als *n* Zeichen, dann liest *...scanf* alle Zeichen des Felds und fährt dann mit dem nächsten Eingabefeld und der nächsten Formatierungsanweisung fort.

Auswirkungen hat eine derartige Angabe nur dann, wenn das entsprechende Feld nicht aufgrund einer anderen Bedingung als beendet betrachtet wird (z.B. durch ein »Whitespace-Zeichen oder ein nicht auswertbares Zeichen wie »8« oder »9« bei der Eingabe einer Oktalzahl).

Größen- und Typangaben N, F und h, l, L

Die Größenangaben (*N* und *F*) und die zusätzlichen Typangaben (*h*, *l* und *L*) steuern die Interpretation des zugehörigen Adreßparameters *arg*.

F und *N* legen eine (vom verwendeten Speichermodell abweichende) Zeigergröße fest.

Die Größenangaben *h*, *l* und *L* legen fest, welcher Typ für die folgenden Eingabedaten verwendet werden soll (*h* = **short** *l* = **long**, *L* = **long double**). Die Eingabedaten werden in den angegebenen Typ umgewandelt. Der jeweilige Adreßparameter *arg* sollte auf ein Objekt mit entsprechender Größe zeigen.

Angabe	Wirkung
F	Das korrespondierende Adreßargument *arg* wird als **far**-Zeiger interpretiert.
N	Das korrespondierende Adreßargument *arg* wird als **near**-Zeiger interpretiert.
h	Für die Formate *d*, *i*, *o*, *u*, und *x*: Konvertierung der Eingabe in das Format **short int** und Speicherung in einem **short-**Objekt.
	Für alle anderen Formate ohne Wirkung.
l	Für die Formate *d*, *i*, *o*, *u*, und *x*: Konvertierung der Eingabe in das Format **long int** und Speicherung in einem **long-**Objekt.
	Für die Formate *e*, *f* und *g*: Konvertierung der Eingabe in das Format **double** und Speicherung in einem **double-**Objekt.
	Für alle anderen Formate ohne Wirkung.
L	Für die Formate *e*, *f* und *g*: Konvertierung der Eingabe in das Format **long double** und Speicherung in einem entsprechenden Objekt.
	Für alle anderen Formate ohne Wirkung.

Abbruch einer Auswertung

scanf kann aus verschiedenen Gründen die Auswertung eines Feldes vor Erreichen des normalen Feldendes (Whitespace-Zeichen) abbrechen oder sogar den Lesevorgang komplett beenden.

scanf beendet die Auswertung des aktuellen Eingabefelds, wenn

- direkt auf das %-Zeichen einer Formatierungsanweisung ein * folgt (Unterdrückung der Zuweisung). Das aktuelle Feld wird zwar vollständig ausgewertet, das Ergebnis aber nicht gespeichert;
- die durch Breitenangabe für dieses Feld angegebene Anzahl von Zeichen erreicht ist;
- das nächste Zeichen im aktuellen Format nicht auswertbar ist (wie beispielsweise ein »A« bei einem dezimalen Format);
- das nächste Zeichen nicht in der Suchmenge bzw. in der invertierten Suchmenge enthalten ist.

Wenn *scanf* aus einem der obigen Gründe die Auswertung des aktuellen Eingabefelds abbricht, dann wird das nächste Zeichen wieder zurückge-

stellt und als das erste Zeichen des folgenden Eingabefeldes bzw. als erstes Zeichen einer folgenden Leseoperation aufgefaßt.

Die Funktion *scanf* endet, wenn eine der folgenden Bedingungen zutrifft:

- Das nächste Zeichen des Eingabefelds entspricht nicht dem erwarteten (da im Formatstring vereinbarten) normalen Zeichen. Der Aufruf `scanf("%d-%d", &a, &b)` erwartet beispielsweise, daß die beiden Integerwerte durch einen Bindestrich getrennt werden. Jedes andere Trennzeichen führt zum Abbruch von *scanf*.
- Das nächste Zeichen des aktuellen Eingabefeldes ist EOF.
- Es wurden so viele Eingabefelder gelesen und bearbeitet, wie durch den Formatstring angegeben werden.

Zeichenfolgen innerhalb des Formatstrings, die nicht Teil einer Formatierungsanweisung sind, müssen in den Eingabedaten exakt so erscheinen und werden von *scanf* ausgewertet, aber nicht gespeichert. Stimmt ein gelesenes Zeichen nicht mit dem erwarteten Zeichen überein, so wird *scanf* abgebrochen. Das Zeichen verbleibt im Eingabefeld, so als wäre es nie gelesen worden.

Rückgabewert

scanf liefert die Anzahl der fehlerfrei gelesenen, konvertierten und gespeicherten Felder zurück, wobei mit * unterdrückte Felder nicht gezählt werden. Wenn überhaupt keine Zuweisungen erfolgt sind, ist der Rückgabewert 0. Der Versuch, mit *scanf* über das Ende einer Datei bzw. über das Ende eines Strings hinauszulesen, ergibt den Wert EOF.

Siehe auch

atof, *cscanf*, *fscanf*, *freopen*, *getc*, *printf*, *sscanf*, *vfscanf*, *vscanf*, *vsscanf*

_searchenv — stdlib.h

Funktion

Sucht nach einer Datei im Verzeichnispfad einer Umgebungsvariablen.

Syntax

```
void _searchenv(const char *file, const char *varname char *buf);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

_searchenv durchsucht zuerst das aktuelle Verzeichnis nach einer Datei mit dem durch den Parameter *file* angegebenen Namen. Bleibt die Suche erfolglos, dann liest die Funktion die Pfadangaben in der Umgebungsvariable *varname* des Betriebssystems und durchsucht sämtliche darin enthaltenen Verzeichnisse der Reihe nach (typische Umgebungsvariablen sind PATH, LIB oder INCLUDE).

Wenn die Datei gefunden wurde, wird der vollständige Dateiname (Laufwerk, Verzeichnis und Dateiname) im durch den Parameter *buf* angegebe-

S–T

nen Puffer gespeichert. Dieser String kann beispielsweise bei einem folgenden Aufruf von *fopen* oder *exec...* verwendet werden. Der Puffer muß groß genug sein, um jeden möglichen Dateinamen aufzunehmen. Kann die Datei nicht gefunden werden, zeigt *buf* auf einen leeren String (der nur ein Nullzeichen enthält).

Rückgabewert Keiner.

Siehe auch *_dos_findfirst*, *_dos_findnext*, *exec...*, *spawn...*, *system*

searchpath — dir.h

Funktion Sucht im Verzeichnispfad des Betriebssystems nach einer Datei.

Syntax

```
char *searchpath(const char *file);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *searchpath* durchsucht zuerst das aktuelle Verzeichnis nach einer Datei mit dem durch den Parameter *file* angegebenen Namen. Bleibt die Suche erfolglos, dann liest die Funktion die Pfadangaben in der Umgebungsvariablen PATH und durchsucht sämtliche dort angegebenen Verzeichnisse der Reihe nach.

Wenn die Datei gefunden wurde, liefert *searchpath* einen Zeiger auf den vollständigen Dateinamen (Laufwerk, Verzeichnis und Dateiname) zurück. Dieser String kann beispielsweise für einen folgenden Aufruf von *fopen* oder *exec...* verwendet werden.

Der zurückgelieferte Zeiger zeigt auf einen statischen Puffer, der bei jedem Aufruf von *searchpath* überschrieben wird.

Rückgabewert Bei erfolgreicher Ausführung wird ein Zeiger auf einen String mit dem gefundenen Dateinamen, im Fehlerfall NULL zurückgeliefert.

Siehe auch *exec...*, *findfirst*, *findnext*, *spawn...*, *system*

_searchstr — stdlib.h

Funktion Durchsucht eine Verzeichnisliste nach einer Datei.

Syntax `void _searchstr(const char *file, const char *ipath, char *buf);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_searchstr* durchsucht zuerst das aktuelle Verzeichnis nach einer Datei mit dem durch den Parameter *file* angegebenen Namen. Bleibt die Suche erfolglos, dann liest die Funktion die Pfadangaben in dem durch den Parameter *ipath* angegebenen String und durchsucht sämtliche darin enthaltenen Verzeichnisse der Reihe nach. Die Verzeichnisse in *ipath* müssen durch Semikolons getrennt sein.

Wenn die Datei gefunden wurde, wird der vollständige Dateiname (Laufwerk, Verzeichnis und Dateiname) im durch den Parameter *buf* angegebenen Puffer gespeichert. Dieser String kann beispielsweise bei einem folgenden Aufruf von *fopen*, oder *exec...* verwendet werden. Der Puffer muß groß genug sein, um jeden möglichen Dateinamen aufzunehmen. Die in stdlib.h definierte Konstante _MAX_PATH bestimmt die Maximallänge eines Dateinames. Kann die Datei nicht gefunden werden, zeigt *buf* auf einen leeren String (der nur ein Nullzeichen enthält).

Rückgabewert Keiner.

Siehe auch *_searchenv*

segread — dos.h

Funktion Liest die Segmentregister des Prozessors.

Syntax `void segread(struct SREGS *segp);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *segread* füllt eine Struktur des Typs SREGS mit den aktuellen Werten der Segmentregister und ist hauptsächlich für folgende Aufrufe von *intdosx* und *int86x* gedacht.

Rückgabewert Keiner.

Siehe auch *FP_OFF*, *int86*, *int86x*, *intdos*, *intdosx*, *MK_FP*, *movedata*

setbuf — stdio.h

Funktion Explizite Zuordnung eines Puffers zu einem Stream.

Syntax

```
void setbuf(FILE *stream, char *buf);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

setbuf ordnet der durch *stream* angegebenen Datei den Puffer *buf* anstelle des automatisch vergebenen Puffers zu. Die Minimalgröße von *buf* ist durch die in stdio.h definierte Konstante BUFSIZ festgelegt. Die Funktion kann nur angewendet werden, wenn die Datei bereits offen ist.

Die Angabe des Zeigerwertes NULL für *buf* bewirkt, daß folgende Ein-/Ausgaben über *stream* ungepuffert arbeiten, d.h. Ausgaben geschehen sofort, Eingaben werden direkt vom entsprechenden Gerät gelesen.

stdin und *stdout* arbeiten ungepuffert, wenn sie nicht umgeleitet sind. Umleitungen bewirken dagegen normalerweise eine Pufferung, die mit *setbuf* verändert werden kann.

Die Pufferung von Ein- und Ausgaben bedeutet: Ausgaben werden zeichenweise zwischengespeichert und blockweise geschrieben; Eingaben werden blockweise von der Datei bzw. dem Gerät gelesen und danach zeichenweise aus dem Puffer gelesen.

setbuf darf nur direkt nach dem Öffnen (oder direkt nach einem *fseek*-Aufruf) auf einen Stream angewendet werden, da die Ergebnisse ansonsten unvorhersehbar sind. Die Anwendung dieser Funktion auf ungepufferte Dateien ist dagegen problemlos und jederzeit möglich.

Für *buf* kann eine lokale Variable verwendet werden. Man sollte allerdings in diesem Fall streng darauf achten, die Datei vor dem Ende der entsprechenden Funktion wieder zu schließen.

Rückgabewert

Keiner.

Siehe auch

fflush, *fopen*, *fseek*, *setvbuf*

setcbrk — dos.h

Funktion

Legt fest, bei welchen Operationen auf `[Strg][Untbr]` geprüft wird.

Syntax

```
int setcbrk(int cbrkvalue);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung

setcbrk schaltet mit Hilfe der DOS-Funktion 0x33 die Überprüfung von `[Strg][Untbr]` ein bzw. aus.

crbkvalue = 0	Überprüfung ist *ausgeschaltet*. Ausnahme: Bei Ein- und Ausgaben über Tastatur, Bildschirm, serielle Schnittstellen und Drucker.
crbkvalue = 1	Überprüfung ist *eingeschaltet*. Bei jedem Systemaufruf wird geprüft, ob [*Strg*][*Untbr*] betätigt wurde.

Rückgabewert *setcbrk* liefert den als *cbrkvalue* übergebenen Wert zurück.

Siehe auch *getcbrk*

_setcursortype conio.h

Funktion Legt die Form des Cursors fest.

Syntax

```
void _setcursortype(int cur_t);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung Folgende Werte sind für *cur_t* möglich:

_NOCURSOR	Unsichtbar
_NORMALCURSOR	Unterstrich (Standard)
_SOLIDCURSOR	Ausgefülltes Kästchen

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

setdate

Siehe *_dos_getdate*.

setdisk

Siehe *getdisk*.

setdta dos.h

S–T

Funktion Setzt die *disk transfer address* (DTA) von DOS.

Syntax

```
void setdta(char far *dta);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				

Beschreibung *setdta* setzt die Startadresse der DTA auf die über *dta* angegebene Adresse.

Rückgabewert Keiner.

Siehe auch *getdta*

setftime

Siehe *getftime*.

setjmp setjmp.h

Funktion Bereitet ein nicht-lokales *goto* vor.

Syntax

```
int setjmp(jmp_buf jmpb);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *setjmp* speichert den Zustand der aktuellen Task in dem durch *jmpb* angegebenen Puffer und gibt 0 zurück.

Ein Aufruf von *longjmp* zu einem späteren Zeitpunkt stellt diesen Zustand wieder her – die Task setzt die Arbeit an der Stelle fort, an der sich der Aufruf von *setjmp* befindet.

Ein Task-Zustand beinhaltet:

Win16	Win32
alle Segmentregister (CS, DS, ES, SS)	Segmentregister werden nicht gesichert
Registervariablen (DI, SI)	Registervariablen (EBX, EDI, ESI)
Stack-Zeiger (SP)	Stack-Zeiger (ESP)
Frame-Zeiger (BP)	Frame-Zeiger (EBP)
Flags	Flags werden nicht gesichert

Ein Task-Status ist vollständig genug, um mit *setjmp* und *longjmp* Koroutinen zu implementieren.

setjmp muß vor *longjmp* aufgerufen werden. Die Routine, die *setjmp* aufgerufen und *jmpb* gesetzt hat, muß immer noch aktiv sein, bevor *longjmp* aufgerufen wird. Andernfalls sind die Ergebnisse unvorhersehbar.

setjmp ist besonders zur Behandlung von Fehlern und Exceptions geeignet, die in maschinennahen Unterprogrammen auftreten.

Rückgabewert *setjmp* liefert beim ersten Aufruf zum Setzen eines Sprungziels immer den Wert 0 zurück. Der »Aufruf« von *setjmp* über *longjmp* setzt den Rückgabewert von *setjmp* auf einen Wert ungleich Null.

Siehe auch *longjmp*, *signal*

setlocale — locale.h

Funktion Auswahl oder Abfrage einer Locale-Einstellung.

Syntax

```
char *setlocale(int category, char *locale);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung Borland C++ unterstützt zur Zeit die folgenden Locale-Einstellungen:

Modul	unterstützte Locale-Einstellung
de_DE	Deutsch
dr_FR	Französisch
en_GB	Englisch (Großbritannien)
en_US	Englisch (USA)

Zukünftige Versionen von Borland C++ werden zusätzliche Locale-Einstellungen unterstützen.

In jeder Locale-Einstellung werden die folgenden Zeichensätze unterstützt:

DOS437	Englisch
DOS850	Mehrsprachig (Latin I)
WIN1252	Windows, mehrsprachig

Eine Beschreibung der DOS-Zeichensätze finden Sie im *MS-DOS Benutzer- und Referenzhandbuch*. Der Zeichensatz WIN1252 wird in *MS Windows 3.1 Programmer's Reference, Volume 4* erläutert.

Folgende Werte sind für den Parameter *category* möglich:

S–T

Wert	Auswirkung
LC_ALL	Auswirkung auf alle folgenden Kategorien
LC_COLLATE	Auswirkung auf *strcoll* und *stxfrm*
LC_CTYPE	Funktionen die mit 1-Byte-Zeichen arbeiten. Keine Auswirkung auf die Funktionen *mbstowcs* und *mbtowc.*
LC_MONETARY	Auswirkung auf die Formatierung von Währungsangaben mit der Funkton *localeconv.*
LC_NUMERIC	Auswirkung auf den Dezimalpunkt bei numerischen Angaben (keine Währungsangaben). Dazu gehören die Funktionsfamilie *printf* und die von *localeconv* zurückgelieferten Informationen.
LC_TIME	Auswirkung auf *strftime*

Der Parameter *locale* enthält einen Zeiger auf den Namen der Locale-Einstellung oder der Locale-Kategorie. Bei Übergabe eines NULL-Zeigers wird die aktuell gültige Locale-Einstellung zurückliefert. Wird ein Zeiger auf einen leeren String übergeben, so prüft *setlocale* anhand von Umgebungsvariablen, welche Locale-Einstellung ausgewählt werden soll.

Die Datei LOCALE.BLL wird im Verzeichnis BC4\BIN installiert.

Wenn Sie eine andere Locale-Einstellung als die Standardeinstellung »C« angeben, versucht *setlocale* die Locale-Daten aus der Locale-Bibliotheksdatei LOCALE.BLL zu lesen. Folgendermaßen wird dabei versucht, auf die Datei zuzugreifen:

1. Das Verzeichnis, in dem sich die ausführbaren Dateien der aktuellen Anwendung befinden, wird durchsucht.
2. Das aktuelle Standardverzeichnis wird durchsucht.
3. Die Umgebungsvariable PATH wird ausgewertet und jedes dort angegebene Verzeichnis wird durchsucht.

setlocale endet, wenn die Locale-Bibliothek nicht gefunden wird.

Wenn die angeforderte Locale-Einstellung nicht hergestellt werden kann, so bleibt die bisherige Einstellung gültig und ein NULL-Zeiger wird zurückgeliefert.

Bei Übergabe eines NULL-Zeigers im Parameter *locale* wird der Locale-String der entsprechenden Kategorie zurückgegeben. Enthält *category* den Wert LC_ALL, dann wird ein vollständiger Locale-String zurückgeliefert. Die Struktur des vollständigen Locale-Strings besteht aus den durch Semikolons getrennten Namen aller Kategorien der aktuellen Locale-Einstellung. Dieser String kann wiederum für den Parameter *locale* verwendet werden, wenn *setlocale* mit LC_ALL aufgerufen wird. Dadurch werden alle im Locale-String angegebenen Kategorien wieder eingesetzt. Locale--Einstellungen können auf diese Weise gespeichert und wiederhergestellt werden. Wird der vollständige Locale-String zusammen mit nur einer Ka-

tegorie benutzt, beispielsweise mit LC_TIME, so wird nur diese Kategorie wiederhergestellt.

In ANSI C wird festgesetzt, daß eine implementationsabhängige Locale-Einstellung verwendet wird, wenn durch den Parameter *locale* ein leerer String übergeben wird. *setlocale* versucht in diesem Fall anhand von entsprechenden Umgebungsvariablen die Locale-Einstellung festzustellen, wie es auch POSIX vorschlägt.

Existiert die Umgebungsvariable LC_ALL, so wird die Kategorie anhand dieser Variable bestimmt. Andernfalls wird die Umgebungsvariable mit demselben Namen wie die angeforderte Locale-Kategorie gesucht und diese bei erfolgreicher Suche entsprechend gesetzt.

Führt alles obige nicht zum Erfolg, so wird die Umgebungsvariable LANG verwendet. Existiert auch diese nicht, dann wird *setlocale* abgebrochen und liefert einen NULL-Zeiger zurück.

Im Kapitel 4 des *Programmierhandbuchs* finden Sie Informationen über die Definition von Optionen.

Sie müssen __USELOCALES__ in jedem Modul definieren, in dem Sie dynamisch ladbare Locale-Einstellungen verwenden wollen. Andernfalls verwenden alle Funktionen und Makros lediglich die standardmäßige Locale-Einstellung »C«.

setlocale liefert die aktuell gültige Locale-Einstellung zurück, wenn im Parameter *locale* ein NULL-Zeiger übergeben wird. Bezeichnet der Parameter *category* eine einzelne Kategorie, wie etwa LC_COLLATE, so enthält der zurückgelieferte String den Namen dieser Kategorie. LC_ALL wird als Parameter *category* verwendet, um den vollständigen Locale-String zu erhalten, der alle aktuell gültigen Kategorienamen enthält.

```
.
.
.
localenameptr = setlocale( LC_COLLATE, NULL );

if (localenameptr)
    printf( "%s\n", localenameptr );
.
.
.
```

Die vom obigen Quelltext erzeugte Ausgabe beinhaltet einen der Modulnamen, zusammen mit der angegebenen Code-Seite, zum Beispiel
`de_DE.DOS850@dbase`

S–T

```
.
.
.
localenameptr = setlocale( LC_ALL, NULL );

if (localenameptr)
    printf( "%s\n", localenameptr );
.
.
.
```

Eine Beispielausgabe des obigen Quelltextes könnte lauten:

```
de_DE.DOS850@dbase;de_DE.DOS850;de_DE.DOS850;de_DE.DOS850;
de_DE.DOS850;de_DE.DOS850;
```

Jede Kategorie in dem obigen, vollständigen Locale-String wird durch ein Semikolon begrenzt. Der String kann von einer Anwendung kopiert und gespeichert werden. Zu einem späteren Zeitpunkt kann dann mit diesem String dieselbe Locale-Einstellung wiederhergestellt werden. Jeder Kategoriename entspricht den in locale.h definierten Konstanten für die Locale-Kategorien. Der erste Name bezeichnet daher die Kategorie LC_COLLATE, der zweite die Kategorie LC_CTYPE und so weiter. Alle weiteren in locale.h definierten Kategorien sind für zukünftige Implementierungen reserviert.

Es folgen einige Beispiele, wie Locale-Einstellungen mit *setlocale* ausgewählt werden können:

Alle Standard-Kategorien für die deutsche Locale-Einstellung setzen:

```
setlocale( LC_ALL, "de_DE.DOS850" );
```

Die standardmäßige Einstellung der Kategorie LC_COLLATE lautet *dbase*. Daher ist es unerheblich, ob Sie *dbase* angeben, oder aber gar nichts – das Ergebnis ist in beiden Fällen gleich. In zukünftigen Versionen kann jedoch eine andere Standardeinstellung enthalten sein.

Die deutsche Locale-Einstellung der Kategorie LC_COLLATE auf *dbase* setzen:

```
setlocale( LC_COLLATE, "de_DE.DOS850@dbase );
```

Wird kein Kategoriename angegeben, dann wird aus der Locale-Bibliothek die Standardkategorie geladen: So lädt beispielsweise der Aufruf:

```
setlocale( LC_COLLATE, "de_DE.DOS850" )
```

die LC_COLLATE-Standardkategorie. Diese kann einen speziellen Namen haben oder auch nicht.

```
setlocale( LC_COLLATE, "de_DE.DOS850@dbase" )
```

lädt die LC_COLLATE-Kategorie `dbase`. Dies kann die Standardkategorie sein oder auch nicht.

setlocale aktualisiert die Locale-Struktur *lconv*, wenn eine Anforderung durchgeführt worden ist.

Beim Beenden der Anwendung wird der für das Locale-Objekt reservierte Speicher wieder freigegeben.

Rückgabewert Bei erfolgreicher Ausführung liefert *setlocale* einen String, der die Locale-Kategorie (oder eventuell alle Kategorien) der neuen Locale-Einstellung bezeichnet.

Im Fehlerfall wird ein NULL-Zeiger zurückgeliefert und die Locale-Einstellung bleibt unverändert. Alle weiteren möglichen Rückgabewerte wurden im Abschnitt **Beschreibung** bereits erläutert.

Siehe auch *localeconv*

setmem mem.h

Funktion Füllt einen Speicherbereich mit einem Wert.

Syntax

```
void setmem(void *dest, unsigned length, char value);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *setmem* setzt den durch *dest* angegebenen Speicherblock der Länge *length* Bytes auf den Wert *value*.

Rückgabewert Keiner.

Siehe auch *memset*, *strset*

setmode fcntl.h

Funktion Setzt den Übersetzungsmodus einer offenen Datei.

Syntax

```
int setmode(int handle, int amode);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

S–T

Beschreibung *setmode* setzt den Modus der durch *handle* bezeichneten Datei entweder auf Text- oder auf Binärmodus. Für *amode* kann genau eine der beiden in fcntl.h definierten Konstanten verwendet werden: O_BINARY oder O_TEXT.

Rückgabewert Bei fehlerfreier Ausführung liefert *setmode* den früheren Übersetzungsmodus zurück; ansonsten ist das Funktionsergebnis -1 und *errno* wird auf den folgenden Wert gesetzt:

EINVAL Ungültiges Argument

Siehe auch *creat*, *open*, *_rtl_creat*, *_rtl_open*

settime

Siehe *gettime*.

setvbuf stdio.h

Funktion Explizite Zuordnung eines Puffers zu einem Stream.

Syntax

```
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *setvbuf* ordnet der durch *stream* angegebenen Datei den Puffer *buf* anstelle des automatisch vergebenen Puffers zu. Die Funktion kann nur angewendet werden, wenn die Datei bereits offen ist.

Mit der Angabe des Zeigerwertes NULL für *buf* wird über einen Aufruf von *malloc* ein Puffer der Größe *size* reserviert und der Datei zugeordnet. Der Puffer wird automatisch freigegeben, wenn die Datei wieder geschlossen wird. Der Parameter *size* bestimmt die Puffergröße und muß größer als 0 sein.

➡ *size* darf nicht größer sein als die in limits.h definierte Konstante UINT_MAX.

stdin und *stdout* arbeiten ungepuffert, solange sie nicht umgeleitet sind. Umleitungen bewirken dagegen vollständige Pufferung. *Ungepuffert* bedeutet, daß Zeichen, die in einen Stream ausgegeben werden sollen, unmittelbar in die Datei bzw. das Gerät geschrieben werden. *Gepuffert* bedeutet hingegen, daß die auszugebenden Zeichen zunächst gesammelt und dann blockweise geschrieben werden.

Der Parameter *type* muß den Wert einer der drei folgenden Konstanten enthalten:

- _IOFBF Die Datei wird vollständig gepuffert. Wenn der Puffer leer ist, versucht der nächste Eingabebefehl, den Puffer komplett mit Daten zu füllen (liest also *size* Bytes). Bei Ausgaben wird

zunächst der Puffer gefüllt, erst danach wird der komplette Pufferinhalt geschrieben.

- _IOLBF Ausgaben werden zeilenweise gepuffert: Der Pufferinhalt wird nach jeder Ausgabe eines Zeilenvorschubs physikalisch geschrieben. Eingaben arbeiten dagegen genauso wie bei vollständiger Pufferung.
- _IONBF Die Datei ist ungepuffert. Die Parameter *buf* und *size* werden ignoriert. Jeder Eingabebefehl liest direkt von der Datei, jeder Ausgabebefehl hat eine sofortige Schreibaktion zur Folge.

Für *buf* kann eine lokale Variable verwendet werden – allerdings sollte man in diesem Fall darauf achten, die Datei vor dem Ende der entsprechenden Funktion wieder zu schließen.

Rückgabewert *setvbuf* liefert bei fehlerfreier Ausführung den Wert 0. Ungültige Werte für *type* oder *size*, fehlender Speicherplatz für *malloc* oder die Anwendung auf eine nicht geöffnete Datei erzeugen ein Funktionsergebnis ungleich null.

Siehe auch *fflush*, *fopen*, *setbuf*

setvect

Siehe *getvect*.

setverify dos.h

Funktion Legt fest, ob Schreiboperationen vom Betriebssystem überprüft werden oder nicht.

Syntax

```
void setverify(int value);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■				■

Beschreibung *setverify* setzt den aktuellen Zustand des Verify-Flag auf den Inhalt des Parameters *value*, der 0 (Überprüfung aus) oder 1 (Überprüfung an) annehmen kann.

Das Verify-Flag beeinflußt Schreiboperationen auf Diskette/Festplatte. Es bestimmt, ob Schreiboperationen auf erfolgreiche Durchführung getestet werden oder nicht.

Rückgabewert Keiner.

Siehe auch *getverify*

S–T

signal — signal.h

Funktion Legt fest, wie das Programm auf Signale reagiert.

Syntax

```
void (_USERENTRY *signal(int sig, void (_USERENTRY *func)
                               (int sig[,int subcode])))(int);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Über die Funktion *signal* können Sie festlegen, wie Ihr Programm auf die Auslösung des Signals *sig* reagiert. Sie können eine eigene Routine zur Bearbeitung des Signals (einen Signal-Handler) definieren oder auch eine der beiden in signal.h vordefinierten Routinen, SIG_DEFL und SIG_IGN einsetzen. Die Funktion *func* muß mit der Aufrufkonvention _USERENTRY benutzt werden.

Funktionszeiger	Bedeutung
SIG_DFL	Das Programm wird beendet.
SIG_ERR	Zeigt an, daß *signal* mit einem Fehler zurückgekehrt ist.
SIG_IGN	Dieser Signaltyp wird ignoriert.

In der folgenden Tabelle finden Sie Signaltypen zusammen mit den von ihnen ausgelösten Standardaktionen:

Signaltyp	Bedeutung
SIGBREAK	Tastatur muß sich im Modus »raw« (Zeichen nicht interpretieren) befinden.
SIGARBRT	Anormales Ende. Standardaktion entspricht dem Aufruf von _exit(3).
SIGFPE	Arithmetischer Fehler, ausgelöst von Division durch Null, ungültiger Operation, usw. Standardaktion entspricht Aufruf von _exit(1).
SIGILL	Ungültige Operation. Standardaktion entspricht Aufruf von _exit(1).
SIGINT	[*Strg*][*C*] Interrupt. Standardaktion: Aufruf des Interrupts 0x23.
SIGSEGV	Ungültiger Speicherzugriff. Standardaktion entspricht Aufruf von _exit(1).
SIGTERM	Beendigung des Programms wird angefordert. Standardaktion entspricht Aufruf von _exit(1).
SIGUSR1	Benutzerdefinierte Signale, die nur durch den Aufruf von *raise* erzeugt werden
SIGUSR2	können. Standardaktion: Das Signal wird ignoriert.
SIGUSR3	

signal.h definiert einen Datentyp *sig_atomic_t* als »Atom« eines Speicherzugriffs, der auch durch asynchrone Interrupts nicht unterbrochen werden kann. Für die Prozessoren der 8086-Familie ist dies ein Wort der

Länge 16 Bit, für 80386 und höher ist es ein Wort der Länge 32Bit, also ein Borland C++ Integer.

Wenn ein Signal durch die Funktion *raise* oder ein externes Ereignis erzeugt wird, dann geschieht folgendes:

- Wurde für das Signal ein benutzerdefinierter Signal-Handler installiert, dann wird die Standardaktion für diesen Signaltyp SIG_DFL gesetzt.
- Der benutzerdefinierte Handler wird aufgerufen. Der Signaltyp wird dabei als Parameter übergeben.

Benutzerdefinerte Handler können mit einem Aufruf von *abort*, *_exit*, *exit* oder *longjmp* enden – oder auch mit einem normalen Rücksprung zum unterbrochenen Programm. Soll Ihre Handler-Funktion weitere Signale empfangen und verarbeiten, so müssen Sie *signal* innerhalb der Handler-Funktion erneut aufrufen.

Borland C++ implementiert einen Erweiterung von ANSI C bei den Signaltypen SIGFPE, SIGSEGV und SIGILL. Die entsprechenden, benutzerdefinierten Signal-Handler bekommen nicht nur den Signaltyp, sondern auch noch einen oder zwei zusätzliche Parameter übergeben. Wenn diese drei Signaltypen über *raise* ausgelöst werden, erhält der Handler einen der in folgender Tabelle aufgeführten Werte als zusätzlichen Parameter (und kann daran erkennen, daß er explizit aufgerufen wurde). Die Werte der folgenden Konstanten sind in float.h deklariert.

Signal	2. Parameter
SIGFPE	FPE_EXPLICITGEN
SIGSEGV	SEGV_EXPLICITGEN
SIGILL	ILL_EXPLICITGEN

Wird ein Signal des Typs SIGFPE als Folge eines Gleitkommafehlers ausgelöst, bekommt der benutzerdefinierte Handler die Art dieses Fehlers als zweiten Parameter übergeben. Die dazugehörigen Konstanten (FPE_...) sind ebenfalls in float.h deklariert. Für die Signaltypen SIGSEGV und SIGILL oder den Typ SIGFPE als Folge eine Integerfehlers erhält der Handler zwei zusätzliche Parameter:

- Den Signaltyp entsprechend dem ANSI-Standard (also SIGSEGGV, SIGILL oder SIGFPE).
- Einen Integer-Zeiger in den Stack der *interrupt*-Routine, die den benutzerdefinierten Signal-Handler aufgerufen hat. Dieser Zeiger zeigt auf eine Liste der Prozessorregister, die beim Auftreten des Fehlers gespeichert wurden. Die Register sind in derselben Reihenfolge wie die Parameter der Interrupt-Funktion (BP, DI, SI, DS, ES, DX, CX, BX, AX, IP, CS, FLAGS) gespeichert. Wenn der Signal-Handler beispielsweise als

```
void Handler(int sig, int type, int *reglist);
```

deklariert ist, dann kann er mit

```
*(reglist +2) = new_DI_value
```

festlegen, welchen Wert das Register DI bei der Rückkehr zum unterbrochenen Programm haben soll (vgl. Beispiel 2).

Für Signale des Typs SIGFPE sind in float.h folgende Konstanten definiert. Sie entsprechen den Fehlern, die der 8087-Prozessor erkennen kann, sowie INTEGER DIVIDE BY ZERO und INTERRUPT ON OVERFLOW beim Hauptprozessor.

Konstante	Bedeutung
FPE_INTOVFLOW	INTO-Befehl mit gesetztem OF-Flag
FPE_INTDIV0	Integerdivision durch Null
FPE_INVALID	Ungültiger 8087-Befehl
FPE_ZERODIVIDE	Division durch Null
FPE_OVERFLOW	Überlauf
FPE_UNDERFLOW	Unterlauf
FPE_INEXACT	Ergebnis ist ungenau (Rundungsfehler)
FPE_EXPLICITGEN	Signal durch *raise(*SIGFPE*)* ausgelöst
FPE_STACKFAULT	Über-/Unterlauf des Gleitkomma-Stack

➥ FPE_INTOVFLOW und FPE_INTDIV0 werden durch Integer-Operationen ausgelöst, die restlichen Signale durch Gleitkommaoperationen, d.h. entweder direkt durch den 8087-Prozessor oder durch die Routinen des 8087-Emulators. Ob ein Ausnahmezustand bei Gleitkommaoperationen ein Signal setzt, hängt vom Steuerwort des Koprozessors (bzw. des Emulators) ab, das über die Routine *_control87* verändert werden kann. Denormalisierte Werte und der dazugehörige Ausnahmezustand (*DeNorm*) werden direkt von Borland C++ behandelt und nicht an den Handler übergeben.

Für Signale des Typs SEGSEGV sind die folgenden Konstanten in signal.h definiert:

SEGV_BOUND	ausgelöst durch BOUND-Befehl
SEGV_EXPLICITGEN	ausgelöst durch *raise(*SIGSEGV*)*

Der Prozessorbefehl BOUND ist für die Prozessortypen 8088/86 nicht definiert. Die Prozessoren 186, 286, 386 sowie NEC V kennen diesen Befehl. Auf 80088/86 Prozessoren kann daher der Type SEGV_BOUND des Signals SIGSEGV nicht auftreten. Borland C++ verwendet den Befehl BOUND nicht. Er kann jedoch in Inline-Assemblercode sowie separat assemblierten Routinen verwendet werden.

Für Signale des Typs SIGILL sind die folgenden Konstanten in signal.h definiert:

ILL_EXECUTION	ungültige Operationsanforderung
ILL_EXPLICITGEN	ausgelöst durch *raise(*SIGILL*)*

Die Prozessortypen 8088/86, NEC V20 und V30 lösen bei illegalen Befehlen keinen Interrupt aus – SIGILL kann hier nur durch einen Aufruf von *raise* ausgelöst werden. Die Prozessortypen 186, 286, 386, NEC V40 und NEC V50 können hingegen SIGILL auslösen.

Bei den Signalen SIGFPE, SIGSEGV oder SIGILL ist ein Rücksprung des Signal-Handler im allgemeinen nicht empfehlenswert, wenn der Koprozessorstatus fehlerhaft, ein Divisionergebnis falsch, ein Überlauf entstanden, ein BOUND-Befehl erfolglos oder eine Operation ungültig ist. Ein Rücksprung ist nur dann empfehlenswert, wenn der Signal-Handler die Register so verändert, daß ein sinnvoller Rücksprungkontext existiert *oder* wenn sich anhand des Signaltyps (beispielsweise FPE_EXPLICITGEN, SEG_EXPLICITGEN oder ILL_EXPLICITGEN) erkennen läßt, daß das Signal explizit per *raise* erzeugt wurde. Im allgemeinen sollten Sie jedoch eine Fehlermeldung ausgeben und das Programm mit *_exit*, *exit* oder *abort* beenden. Wird dennoch ein Rücksprung durchgeführt, dann ist die Programmfortführung wahrscheinlich nicht vorhersehbar.

Rückgabewert

Bei fehlerfreier Ausführung liefert *signal* einen Zeiger auf die Routine, die zuvor als Handler für Signale des angegebenen Typs gesetzt war. Im Fehlerfall ist das Ergebnis von *signal* der Zeiger SIG_ERR, *errno* wird auf den Wert EINVAL gesetzt.

Siehe auch

abort, *_control87*, *ctrlbrk*, *exit*, *longjmp*, *raise*, *setjmp*

sin, sinl — math.h

Funktion

Liefert den Sinus des Arguments zurück.

Syntax

```
double sin(double x);
long double sinl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
sin	■	■	■	■	■		■
sinl	■		■	■			■

Beschreibung

sin erwartet einen Winkel in der Einheit *rad* als Argument und liefert den dazugehörigen Sinus zurück.

sinl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück. Die Fehlerbehandlung kann für diese Funktionen über *_matherr* bzw. *_matherrl* modifiziert werden.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert *sin* und *sinl* liefern den Sinus des Eingabewertes.

Siehe auch *acos, asin, atan, atan2, bcd, complex, cos, tan*

sinh, sinhl — math.h

Funktion Liefert den Sinus hyperbolicus des Arguments zurück.

Syntax

```
double sinh(double x);
long double sinhl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
sinh	■	■	■	■	■		■
sinhl	■		■	■			■

Beschreibung *sinh* berechnet den Sinus hyperbolicus. Dieser ist definiert durch $(e^{x} - e^{-x}) / 2$.

sinhl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück. Die Behandlung von Fehlern kann für diese Funktionen über *_matherr* bzw. *_matherrl* modifiziert werden.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert Bei fehlerfreier Ausführung liefern die Funktionen den Sinus hyperbolicus des Arguments. Bei Überläufen liefert *sinh* HUGE_VAL bzw. *sinhl* _LHUGE_VAL (mit korrektem Vorzeichen) zurück und die globale Variable *errno* erhält den Wert:

ERANGE Ergebnis außerhalb des Rechenbereichs

Siehe auch *acos, asin, atan, atan2, bcd, complex, cos, cosh, sin, tan, tanh*

sleep — dos.h

Funktion Hält die Ausführung des Programms für einen bestimmten Zeitraum an.

Syntax

```
void sleep(unsigned seconds);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■			■

Beschreibung *sleep* hält die Ausführung für die durch *seconds* angegebene Anzahl von Sekunden an und verwendet dabei die Uhr des Systems. Die Genauigkeit

der Routine beträgt dabei eine hundertstel Sekunde bzw. die Genauigkeit der Uhr des Betriebssystems, je nachdem, welche weniger genau ist.

Rückgabewert Keiner.

Siehe auch *delay*

sopen — fcntl.h, sys\stat.h, share.h, io.h

Funktion Öffnet eine Datei in einem Modus für gemeinsamen Zugriff.

Syntax

```
int sopen(char *path, int access, int shflag[, int mode]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *sopen* öffnet die durch *path* angegebene Datei und bereitet je nach den für *access*, *shflag* und *mode* angegebenen Werten die Datei für gemeinsame Lese- und/oder Schreibzugriffe vor.

access wird dabei durch die bitweise ODER-Verknüpfung der in den folgenden beiden Tabellen aufgeführten Flags gebildet. Von der ersten Tabelle darf nur ein Flag benutzt werden. Die Flags der zweiten Tabelle dürfen dagegen beliebig kombiniert werden.

1. Flags für Lesen und/oder Schreiben

O_RDONLY	Nur Lesen
O_WRONLY	Nur Schreiben
O_RDWR	Lesen und Schreiben

2. Weitere Zugriff-Flags

O_NDELAY	Wird nicht verwendet und ist nur aus Gründen der UNIX-Kompatibilität definiert.
O_APPEND	Falls gesetzt, wird der Dateizeiger vor jedem Schreibzugriff an das Dateiende gesetzt.
O_CREAT	Falls die Datei bereits existiert, hat dieses Flag keine Wirkung. Wenn die Datei noch nicht existiert, wird sie erzeugt und die Dateiattribute werden – wie beim Aufruf von *chmod* – anhand der in *mode* gesetzten Bits festgelegt.
O_TRUNC	Wenn die Datei bereits existiert, wird ihr vorheriger Inhalt gelöscht. Die Dateiattribute bleiben unverändert.
O_EXCL	Wird nur zusammen mit O_CREAT verwendet. Falls die Datei bereits existiert, wird ein Fehler zurückgeliefert.
O_BINARY	Die Datei wird im Binärmodus geöffnet..
O_TEXT	Die Datei wird im Textmodus geöffnet.
O_NOINHERIT	Die Datei wird nicht an Kindprozesse weitergegeben.

Die symbolischen Konstanten O_... sind in fcntl.h definiert.

Wenn weder O_BINARY noch O_TEXT angegeben sind, wird die Datei in dem Modus, der durch die globale Variable *_fmode* gesetzt ist, geöffnet.

Bei der Verwendung von O_CREAT (d.h. der Erzeugung einer neuen Datei) bei der Bildung von *access* muß das Argument *mode* auf eine der folgenden symbolischen Konstanten gesetzt werden, die in sys\stat.h definiert sind:

Wert von *mode*	Zugriffserlaubnis
S_IWRITE	Schreiben erlaubt
S_IREAD	Lesen erlaubt
S_IREAD \| S_IWRITE	Lesen und Schreiben erlaubt

shflag legt den Typ des gemeinsamen Zugriffs fest, der auf die Datei *path* angewendet werden soll. Die symbolischen Konstanten für *shflag* sind in share.h festgelegt.

Wert von *shflag*	Beschreibung
SH_COMPAT	Setzt Kompatibilitätsmodus
SH_DENYRW	Lesen und Schreiben nicht erlaubt
SH_DENYWR	Schreiben nicht erlaubt
SH_DENYRD	Lesen nicht erlaubt
SH_DENYNONE	Lesen und Schreiben erlaubt
SH_DENYNO	Lesen und Schreiben erlaubt

Rückgabewert

Wenn die Datei fehlerfrei geöffnet werden konnte, liefert die Funktion das Handle dieser Datei zurück (einen Integerwert größer Null, über den weitere Operationen mit dieser Datei ausgeführt werden). Der Dateizeiger wird auf den Dateianfang gesetzt. Im Fehlerfall liefert die Funktion den Wert -1 und setzt die globale Variable *errno* auf einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
EINVACC	Ungültiger Zugriffscode
ENOENT	Pfad/Datei nicht vorhanden
EMFILE	Zu viele Dateien bereits offen

Siehe auch

chmod, *close*, *creat*, *lock*, *lseek*, *open*, *_rtl_open*, *unlock*, *unmask*

spawn... process.h, stdio.h

Funktion

Erzeugung und Ausführung eines Kindprozesses.

Syntax

```
int spawnl(int mode, char *path, char *arg0, *arg1, ..., *argn, NULL);
int spawnle(int mode, char *path, char *arg0, *arg1, ..., *argn, NULL, char *envp[]);
int spawnlp(int mode, char *path, char *arg0, *arg1, ..., *argn, NULL);
int spawnlpe(int mode, char *path, char *arg0, *arg1, ... , *argn, NULL,
             char *envp[]);
int spawnv(int mode, char *path, char *argv[]);
int spawnve(int mode, char *path, char *argv[], char *envp[]);
int spawnvp(int mode, char *path, char *argv[]);
int spawnvpe(int mode, char *path, char *argv[], char *envp[]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

Die Funktionen der *spawn...*-Familie erzeugen Kindprozesse, die ihre eigenen Dateien ausführen. Es muß ausreichend Speicherplatz für das Laden und Ausführen dieser Prozesse vorhanden sein.

Der Wert des Parameters *mode* legt fest, wie sich der aufrufende Prozeß, der sogenannte Elternprozeß, während der Ausführung des Kindprozesses verhält. Die folgenden Konstanten sind in process.h definiert:

P_WAIT	Der Elternprozeß bleibt im Speicher, wird aber für die Dauer der Ausführung des Kindprozesses »angehalten«.
P_NOWAIT	Die Ausführung des Elternprozesses wird nach dem Start des Kindprozesses fortgesetzt. Die Prozeß-ID des Kindprozesses wird zurückgeliefert, so daß der Elternprozeß über eine Aufruf von *cwait* oder *wait* auf das Ende des Kindprozesses warten kann.
P_NOWAITO	Entspricht P_NOWAIT mit der Ausnahme, daß die ID des Kindprozesses vom Betriebssystem nicht gespeichert wird. Daher kann der Elternprozeß nicht mit *cwait* oder *wait* auf das Ende des Kindprozesses warten.
P_DETACH	Entspricht P_NOWAITO mit der Ausnahme, daß der Kindprozeß im Hintergrund ohne Tastatur- und Bildschirmzugriff ausgeführt wird.
P_OVERLAY	Der Elternprozeß wird vom Kindprozeß im Speicher überlagert und nach dessen Beendigung nicht fortgesetzt. Dies entspricht dem Aufruf einer *exec...*-Funktion.

Der Parameter *path* gibt den Namen des Kindprozesses an. Dabei wird der Standard-Suchalgorithmus des Betriebssystems verwendet:

- Wenn weder eine Namenserweiterung noch ein Punkt angegeben ist, wird zuerst nach Dateien ohne Erweiterung, dann nach .EXE-Dateien und anschließend nach .COM-Dateien gesucht. Ist die Suche erfolglos, dann wird .BAT an den Namen angehangen und noch einmal gesucht.

Der Kommandozeileninterpreter COMSPEC wird zur Ausführung der Datei benutzt.

- Wenn eine Namenserweiterung oder ein Punkt angegeben ist, wird nur nach dem angegebenen Dateinamen gesucht.
- Die Varianten von *spawn...* mit der Namensendung *p* suchen, sofern in *path* kein Verzeichnis explizit angegeben ist, zuerst im aktuellen Verzeichnis und danach in sämtlichen Verzeichnissen, die durch die Umgebungsvariable PATH festgelegt sind.

Die einzelnen Funktionen der *spawn...*-Familie unterscheiden sich durch die Namensendungen *p*, *l*, *v*, und *e*, wobei jeder dieser Buchstaben für andere Fähigkeiten steht. Im einzelnen bedeuten:

p Die *spawn*-Funktion sucht nach dem Kindprozeß nicht nur innerhalb des aktuellen Verzeichnisses, sondern auch in sämtlichen durch die Umgebungsvariable PATH festgelegten Verzeichnissen. Die Varianten ohne *p* suchen nur innerhalb des aktuellen Verzeichnisses.

l Die Argumentzeiger werden an den Kindprozeß in Form einer Liste einzelner Argumente übergeben, also als *arg0, arg1, ... argn*. Die Listenform wird normalerweise verwendet, wenn die Zahl der Parameter vorher bekannt (und konstant) ist.

v Die Argumentzeiger werden an den Kindprozeß als Zeiger-Array übergeben, d.h. als *argv*[0], *argv*[1] ...*argv*[n]. Diese Form wird normalerweise verwendet, wenn die Anzahl der Parameter variabel ist.

e Erlaubt die Übergabe des Zeiger-Array *envp* (d.h. eines eigenen Environment für den Kindprozeß). Ein über die *spawn...*-Varianten ohne *e* aufgerufener Kindprozeß verwendet dieselben Umgebungsvariablen wie der Elternprozeß, er »erbt« das Environment des Elternprozesses.

Jede Funktion der *spawn...*-Familie enthält entweder ein *l* oder ein *v* im Namen, die weiteren Kombinationen mit *p* und *e* sind optional. Zwei Beispiele dazu:

- *spawnl* verwendet eine Liste von Argumenten, sucht nur im aktuellen Verzeichnis nach dem Kindprozeß und übergibt das Environment des Elternprozesses an den Kindprozeß.
- *spawnvpe* übergibt die Argumentzeiger als Zeiger-Array, führt nötigenfalls eine Suche nach dem Kindprozeß über die mit PATH gesetzten Verzeichnisse aus und ändert mit dem Parameter *env* das Environment des Kindprozesses.

Sämtliche *spawn...*-Funktionen müssen mindestens ein Argument an den Kindprozeß übergeben, nämlich *arg0* bzw. *argv*[0]. Dieses erste Argument enthält per Konvention den Pfadnamen des Kindprozesses. Die Übergabe anderer Daten als erstem Argument erzeugt jedoch keinen Fehler. Zur

Übergabe einer leeren Argumentliste an den Kindprozeß muß *arg0* bzw. *argv*[0] gleich NULL sein.

Ein in *arg0* bzw. *argv*[0] übergebener *path*-Wert ist nur unter den DOS-Versionen ab 3.0 für den Kindprozeß verfügbar. Bei älteren Versionen kann der Kindprozeß den in *arg0* bzw. *argv*[0] übergebenen Wert nicht benutzen.

Bei Verwendung der *l*-Funktionen zeigt *arg0* normalerweise auf denselben String wie *path*. *arg1,...,argn* zeigen auf weitere Strings, die Argumentliste wird durch ein (obligatorisches!) NULL beendet.

Bei den *e*-Funktionen wird eine Liste mit neuen Environment-Einstellungen mit dem Parameter *envp* übergeben. *envp* zeigt auf ein Array von **char**-Zeigern. Jedes Element dieses Array ist ein nullterminierter String der Form

```
U_Var = Wert
```

wobei *U_Var* der Name der Umgebungsvariable und *Wert* der String-Wert ist, auf den *U_Var* gesetzt wird. Das letzte Element des Array muß einen Zeiger mit dem Wert NULL enthalten, der das Listenende gekennzeichnet. Wenn *envp* den Wert NULL hat, verwendet der Kindprozeß das Environment des Elternprozesses.

Die Gesamtlänge sämtlicher als *arg* übergebener Parameter darf 260 Bytes nicht überschreiten (inklusive der automatisch eingesetzten Leerzeichen zur Trennung einzelner Parameter). Nullzeichen werden bei dieser Zählung nicht berücksichtigt.

spawn... schließt keine Dateien – durch den Elternprozeß geöffnete Dateien stehen auch dem Kindprozeß zur Verfügung.

Rückgabewert

Bei fehlerfreier Ausführung liefert eine *spawn...*-Funktion mit *mode* P_WAIT den Exit-Code des Kindprozesses zurück (0 bei normaler Beendigung). Wenn der Kindprozeß *exit* mit einem Wert ungleich null explizit aufruft, so liefert die Funktion diesen Wert zurück. Ist *mode* auf P_NOWAIT, P_NOWAITO oder P_DETACH gesetzt, dann liefern die *spawn...*-Funktionen die Prozeß-ID des Kindes zurück, die wiederum an *cwait* übergeben werden kann.

Konnte die *spawn...*-Funktion dagegen nicht ausgeführt werden, dann wird -1 zurückgeliefert, und *errno* erhält einen der folgenden Werte:

E2BIG	Argumentliste zu lang
EINVAL	Ungültiger Wert für *mode*
ENOENT	Pfad/Dateiname nicht gefunden
ENOEXEC	*exec*-Formatfehler
ENOMEM	Nicht genug Platz im Hauptspeicher

Siehe auch

abort, *atexit*, *cwait*, *_exit*, *exit*, *exec...*, *_fpreset*, *searchpath*, *system*, *wait*

_splitpath — stdlib.h

Funktion Zerlegt einen vollständigen Pfadnamen in seine Einzelkomponenten.

Syntax

```
void _splitpath(const char *path, char *drive, char *dir, char *name, char *ext);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_splitpath* übernimmt im Parameter *path* einen vollständigen Dateinamen als String der Form

```
X:\DIR\SUBDIR\NAME.EXT
```

und zerlegt ihn in seine vier Einzelkomponenten. Diese werden in den durch *drive*, *dir*, *name* und *ext* bezeichneten Strings gespeichert. Alle fünf Parameter müssen übergeben werden, einzelne davon dürfen aber NULL sein, was zur Folge hat, daß die zugehörige Komponente zwar analysiert, aber nicht gespeichert wird. Für Maximalgrößen aller beteiligen Strings sind in stdlib.h folgende Konstanten definiert, wobei das abschließende Nullzeichen jeweils mitgezählt werden muß:

Konstante	String
_MAX_PATH	*path*
_MAX_DRIVE	*drive* inklusive Doppelpunkt (:)
_MAX_DIR	*dir* inklusive führendem und abschließendem umgekehrten Schrägstrich (\)
_MAX_FNAME	*name*
_MAX_EXT	*ext* inklusive führendem Punkt (.)

_splitpath unterstellt, daß die durch *drive*, *dir*, *name* und *ext* bezeichneten Strings eine ausreichende Größe haben, um jede nicht leere Komponente zu speichern.

_splitpath verteilt die Interpunktionszeichen aus *path* in folgender Weise:

- *drive* enthält den Laufwerksbezeichner mit einem Doppelpunkt (C:, A:, usw.).
- *dir* enthält einen führenden und abschließenden umgekehrten Schrägstrich (\BC4\INCLUDE\, \SOURCE\, usw.).
- *name* enthält den Dateinamen.
- *ext* enthält den Punkt, der die Namenserweiterung einleitet (.C, .EXE, usw.).

_makepath und *_splitpath* verhalten sich invers zueinander. Die Anwendung von *_makepath* auf die durch *_splitpath* ermittelten Elemente eines vollständigen Pfadnamens ergeben wieder das Original.

Rückgabewert Keiner.

Siehe auch *_fullpath*, *_makepath*

sprintf stdio.h

Funktion Formatierte Ausgabe in einen String.

Syntax

```
int sprintf(char *buffer, const char *format[, argument,...]);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *sprintf* übernimmt eine Reihe von Argumenten, auf die jeweils ein Formatierungsanweisung angewendet wird. Die Formatierungsanweisungen sind in dem String enthalten, auf den *format* zeigt. Die so formatierten Daten werden in den durch *buffer* angegebenen String geschrieben.

Weitere Details über Formatierungsanweisungen finden Sie unter *printf*.

sprintf wendet die erste Formatierungsanweisung auf das erste Argument an, die zweite auf das zweite Argument usw. Die Anzahl der Formatierungsanweisungen und die der Argumente muß übereinstimmen.

Rückgabewert *sprintf* liefert die Anzahl der ausgegebenen Zeichen zurück, wobei das abschließende Nullzeichen nicht mitgezählt wird. Im Fehlerfall ist das Funktionsergebnis EOF.

Siehe auch *fprintf*, *printf*

sqrt, sqrtl math.h

Funktion Berechnet die positive Quadratwurzel des Arguments.

Syntax

```
double sqrt(double x);
long double sqrtl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
sqrt	■	■	■	■	■		■
sqrtl	■		■	■			■

Beschreibung *sqrt* berechnet die positive Quadratwurzel des Arguments *x*.

sqrtl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück. Die Fehlerbearbeitung kann für diese Funktionen mit *_matherr* und *_matherrl* modifiziert werden.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

S–T

Rückgabewert *sqrt* und *sqrtl* liefern die positive Quadratwurzel des Arguments, wenn *x* eine reelle Zahl und positiv ist. Ist *x* reell und negativ, wird die globale Variable *errno* auf den Wert

EDOM Bereichsfehler

gesetzt.

Siehe auch *bcd*, *complex*, *exp*, *log*, *pow*

srand stdlib.h

Funktion Initialisiert den Zufallszahlengenerator.

Syntax

```
void srand(unsigned seed);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Der Zufallszahlengenerator wird beim Aufruf von *srand* mit dem *seed* == 1 neu initialisiert. Durch einen Aufruf von *srand* mit einem bestimmten Parameter *seed* kann er auf einen neuen Startwert gesetzt werden.

Rückgabewert Keiner.

Siehe auch *rand*, *random*, *randomize*

sscanf stdio.h

Funktion Liest und formatiert Eingaben aus einem String.

Syntax

```
int sscanf(const char *buffer, const char *format[, address,...];
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *sscanf* liest eine Reihe von Eingabefeldern zeichenweise aus dem durch *buffer* angegebenen String. Auf jedes Feld wird dann die entsprechende Formatierungsanweisung angewendet, die im durch *format* angegebenen Formatstring enthalten ist. Gelesene Eingaben werden an den Adressen gespeichert, die in den Parametern hinter *format* übergeben werden. Die Anzahl von Eingabefeldern muß mit der Anzahl von Formatierungsanweisungen und Adressen übereinstimmen.

Details zu Formatierungsanweisungen finden Sie unter *scanf*.

sscanf kann das Lesen eines bestimmten Eingabefeldes abbrechen, bevor das normale Feldende (ein Whitespace-Zeichen) erreicht ist. *sscanf* kann auch aus verschiedenen Gründen den Lesevorgang vollständig abbrechen. Die möglichen Ursachen hierfür werden unter *scanf* erläutert.

Rückgabewert *sscanf* liefert die Anzahl der fehlerfrei gelesenen, konvertierten und gespeicherten Felder zurück. Gelesene, aber nicht gespeicherte Felder werden nicht mitgezählt. Wurden keine Felder gespeichert, so wird 0 zurückgeliefert.

Der Versuch, über das Ende des Strings hinauszulesen, liefert den Wert EOF.

Siehe auch *fscanf*, *scanf*

stackavail — malloc.h

Funktion Ermittelt die Größe des verfügbaren Speichers.

Syntax `size_t stackavail(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *stackavail* gibt die Größe des verfügbaren Speichers auf dem Stack in Bytes zurück. Diese Größe entspricht dem dynamischen Speicher, auf den *alloca* zugreifen kann.

Rückgabewert *stackavail* liefert einen Wert vom Typ *size_t* zurück, der der Größe des verfügbaren Speichers in Bytes entspricht.

Siehe auch *alloca*

stat

Siehe *fstat*.

_status87 — float.h

S–T

Funktion Liefert den Status der Gleitkommafunktionen zurück.

Syntax `unsigned int _status87(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Diese Funktion liefert den aktuellen Wert des Statusworts der Gleitkommafunktionen zurück. Das Wort muß als Bitmuster interpretiert werden und enthält neben den Statusbits des 80x87 Koprozessors verschiedene Flags, die von der Routine zur Fehlerbehandlung gesetzt werden.

Rückgabewert Die Bits des Rückgabewerts stellen den Gleitkommastatus dar. In der Header-Datei float.h finden Sie die vollständige Definition der von *_status87* zurückgelieferten Bits.

stime time.h

Funktion Setzt Datum und Uhrzeit des Systems.

Syntax `int stime(time_t *tp);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■				■

Beschreibung *stime* setzt Datum und Uhrzeit des Systems auf den über den Zeiger *tp* angegebenen Wert, der in Sekunden seit dem 1. Januar 1970, 0 Uhr 00:00 (Greenwich Mean Time) gemessen wird.

Rückgabewert *stime* liefert immer den Wert 0.

Siehe auch *asctime*, *ftime*, *gettime*, *gmtime*, *localtime*, *time*, *tzset*

stpcpy string.h

Funktion Kopiert einen String in einen anderen.

Syntax `char *stpcpy(char *dest, const char *src);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *stpcpy* kopiert den Inhalt des durch *src* angegebenen nullterminierten Strings in den durch *dest* angegebenen Speicherbereich. Der Vorgang wird abgebrochen, sobald das abschließende Nullzeichen von *src* erreicht wird.

Rückgabewert *stpcpy* liefert einen Zeiger zurück, der auf das neue Ende von *dest* zeigt, also *dest* + *strlen*(*src*).

Siehe auch *strcpy*

strcat, _fstrcat — string.h

Funktion Hängt einen String an einen anderen.

Syntax

```
char *strcat(char *dest, const char *src);
char far * far _fstrcat(char far *dest, const char far *src);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strcat* hängt sämtliche Zeichen von *src* an die bereits in *dest* vorhandenen Zeichen an. Die Länge des resultierenden Strings ist *strlen*(*dest*) + *strlen*(*src*).

Rückgabewert Zurückgeliefert wird ein Zeiger auf den zusammengefügten String.

Siehe auch *_fstr**

strchr, _fstrchr — string.h

Funktion Sucht einen String nach dem ersten Auftreten eines bestimmten Zeichens ab.

Syntax

```
char *strchr(const char *s, int c);                          /* nur C */
char far * far _fstrchr(const char far *s, int c)            /* C und C++/

const char *strchr(const char *s, int c);                     // nur C++
char *strchr( char *s, int c);                                // nur C++
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strchr* sucht den über *s* angegebenen String nach dem Zeichen *c* ab, wobei die Suche mit dem ersten Zeichen von *s* beginnt. *strchr* findet die *erste* Fundstelle des Zeichens *c* im String *s*. Das abschließende Nullzeichen wird zum Stringinhalt gerechnet und als letztes verglichen, so daß ein Aufruf wie

```
strchr(strs,0)
```

einen Zeiger auf das abschließende Nullzeichen von *strs* liefert.

Rückgabewert *strchr* liefert einen Zeiger auf die erste Fundstelle des Zeichens *c* im String *s* zurück bzw. den Wert NULL, wenn der String dieses Zeichen nicht enthält.

Siehe auch *_fstr**, *strcspn*, *strrchr*

strcmp, _fstrcmp — string.h

Funktion Vergleicht zwei Strings miteinander.

Syntax

```
int strcmp(const char *s1, const char *s2);
int far _fstrcmp(const char far *s1, const char far *s2);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strcmp* führt mit den Inhalten von *s1* und *s2* einen **unsigned**-Vergleich durch. Beginnend mit dem ersten Zeichen werden die beiden Strings zeichenweise verglichen, bis zwei korrespondierende Zeichen ungleich sind oder das Ende der Strings erreicht wird.

Rückgabewert *strcmp* liefert einen Wert

- < 0 wenn *s1* kleiner als *s2* ist
- == 0 wenn *s1* gleich *s2* ist
- > 0 wenn *s1* größer als *s2* ist

Siehe auch *_fstr*, strcmpi, strcoll, stricmp, strncmp, strncmpi, strnicmp*

strcmpi — string.h

Funktion Vergleicht zwei Strings ohne Unterscheidung zwischen Groß- und Kleinschreibung.

Syntax

```
int strcmpi(const char *s1, const char *s2);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■			■

Beschreibung *strcmpi* führt mit den Inhalten von *s1* und *s2* einen **unsigned**-Vergleich durch; zwischen Groß- und Kleinbuchstaben wird dabei nicht unterschieden.

➥ *strcmpi* unterscheidet bei deutschen Umlauten dennoch zwischen Groß- und Kleinschreibung.

Die Routinen *stricmp* und *strcmpi* haben dieselbe Wirkung. *strcmpi* ist ein in string.h implementiertes Makro, das Aufrufe von *strcmpi* in Aufrufe von *stricmp* übersetzt. *strcmpi* wurde aus Kompatibilitätsgründen zu anderen Compilern definiert. Für die Verwendung von *strcmpi* muß daher die Header-Datei string.h eingebunden werden.

Rückgabewert *strcmpi* liefert einen Wert

- < 0 wenn *s1* kleiner als *s2* ist
- == 0 wenn *s1* gleich *s2* ist
- > 0 wenn *s1* größer als *s2* ist

Siehe auch *strcmp*, *strcoll*, *stricmp*, *strncmp*, *strncmpi*, *strnicmp*

strcoll — string.h

Funktion Vergleicht zwei Strings.

Syntax

```
int strcoll(char *s1, char *s2);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *strcoll* vergleicht den String, auf den *s1* zeigt, mit dem String, auf den *s2* zeigt. Dabei wird die in der Kategorie LC_COLLATE der aktuellen locale-Einstellung gesetzte Vergleichsreihenfolge verwendet.

Rückgabewert *strcoll* liefert einen Wert

- < 0 wenn *s1* kleiner als *s2* ist
- == 0 wenn *s1* gleich *s2* ist
- > 0 wenn *s1* größer als *s2* ist

Siehe auch *strcmp*, *strcmpi*, *stricmp*, *strncmp*, *strncmpi*, *strnicmp*, *strxfrm*

strcpy, _fstrcpy — string.h

Funktion Kopiert einen String in einen anderen.

Syntax

```
char *strcpy(char *dest, const char *src);
char far *_fstrcpy(char far *dest, const char far *src);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strcpy* kopiert den Inhalt des über *src* angegebenen Strings in den durch *dest* angegebenen Speicherbereich. Das abschließende Nullzeichen von *src* wird als letztes Zeichen kopiert.

Rückgabewert *strcpy* liefert *dest* zurück.

Siehe auch *_fstr**, *stpcpy*

strcspn, _fstrcspn — string.h

Funktion Liefert die Länge des Teilstrings zurück, der *keines* der Zeichen einer bestimmten Zeichenmenge enthält.

Syntax

```
size_t strcspn(const char *s1, const char *s2);
size_t far *far _fstrcspn(const char far *s1, const char far *s2);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strcspn* liest den String *s1* zeichenweise, bis eines der in *s2* enthaltenen Zeichen auftritt. Die Anzahl der in *s1* gelesenen Zeichen bildet den Rückgabewert. Das abschließende Nullzeichen wird nicht mitgezählt. Beide Strings werden durch den Suchvorgang nicht verändert.

Rückgabewert *strcspn* liefert die Länge des Teilstrings von *s1* zurück, der nur aus Zeichen besteht, die in *s2* nicht vorkommen.

Siehe auch *_fstr**, *strchr*, *strrchr*

_strdate — time.h

Funktion Konvertiert das aktuelle Datum in einen String.

Syntax

```
char *_strdate(char *buf);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_strdate* konvertiert das aktuelle Datum in einen String und speichert diesen in dem durch *buf* angegebenen Puffer. *buf* muß mindestens 9 Zeichen lang sein.

Der String ist durch ein Nullzeichen abgeschlossen und hat das Format MM/DD/YY. MM, DD und YY sind Zahlen mit jeweils zwei Ziffern und geben Monat, Tag und Jahr an.

Rückgabewert Der Rückgabewert von *_strdate* ist *buf*, die Adresse des Datum-Strings.

Siehe auch *asctime*, *ctime*, *localtime*, *strftime*, *_strtime*, *time*

strdup, _fstrdup — string.h

Funktion Kopiert einen String in einen neuen Speicherbereich.

Syntax

```
char *strdup(const char *s);
char far * far _fstrdup(const char far *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

strdup bestimmt zuerst die Länge des übergebenen Strings *s*, reserviert danach mit *malloc* einen Speicherbereich entsprechender Größe (*strlen*(*s*) + 1) und kopiert den Inhalt von *s* in diesen neu reservierten Speicherbereich. Der Programmierer muß den Speicher selbst wieder freigeben, wenn dieser nicht mehr benötigt wird.

Rückgabewert

strdup liefert einen Zeiger auf den neu reservierten Speicherbereich, der den duplizierten String enthält, zurück. Falls nicht genug Platz im Speicher ist (d.h. *malloc* einen Fehler liefert), dann ist das Funktionsergebnis von *strdup* NULL.

Siehe auch

free, *_fstr**

_strerror — string.h

Funktion

Generiert einen String mit einer Fehlermeldung.

Syntax

```
char *_strerror(const char *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

Mit der Funktion *_strerror* können eigene Fehlermeldungen generiert werden. *_strerror* liefert einen Zeiger auf einen nullterminierten String zurück, der die Fehlermeldung enthält.

- Wenn für *s* der Wert NULL angegeben wird, dann zeigt der von *_strerror* zurückgelieferte Wert auf die letzte Fehlermeldung.
- Wird für *s* ein Wert ungleich NULL angegeben, dann enthält der generierte String in dieser Reihenfolge den durch den Parameter *s* übergebenen String (Ihre Fehlermeldung), einen Doppelpunkt, ein Leerzeichen, die zuletzt erzeugte Systemfehlermeldung und einen Zeilenvorschub (\n). Der übergebene String *s* darf höchstens 94 Zeichen lang sein.

Rückgabewert

_strerror liefert einen Zeiger auf den generierten Fehlerstring. Der String wird in einem statischen Puffer erzeugt, der bei jedem Aufruf dieser Funktion überschrieben wird.

Siehe auch

perror, *strerror*

S–T

strerror string.h

Funktion Liefert einen Zeiger auf einen String mit einer Fehlermeldung.

Syntax

```
char *strerror(int errnum);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *strerror* erwartet einen **int**-Wert im Parameter *errnum* und liefert einen Zeiger auf einen statischen String zurück, der die zum Wert von *errnum* gehörige Fehlermeldung enthält.

Rückgabewert *strerror* liefert einen Zeiger auf den generierten Fehlerstring. Der String wird in einem statischen Puffer erzeugt, der bei jedem Aufruf dieser Funktion überschrieben wird.

Siehe auch *perror*, *_strerror*

strftime time.h

Funktion Formatiert die Uhrzeit für die Ausgabe.

Syntax

```
size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm *t);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *strftime* formatiert die Uhrzeit im Argument *t* und gibt sie in das Array aus, auf das das Argument *s* zeigt. Bei der Formatierung werden die in *fmt* enthaltenen Formatierungsanweisungen angewandt. Der Formatstring besteht aus null oder mehr Anweisungen und normalen Zeichen. Wie bei *printf* besteht eine Anweisung aus dem Zeichen %, gefolgt von dem Zeichen für das einzusetzende Format. Alle normalen Zeichen werden unverändert kopiert. Es werden maximal *maxsize* Zeichen nach *s* geschrieben.

Die Uhrzeit wird entsprechend der Kategorie LC_TIME der aktuellen locale-Einstellung formatiert.

In der folgenden Tabelle werden die von ANSI definierten Formatangaben aufgeführt.

Formatierungsanweisung	steht für
%%	Prozentzeichen %
%a	abgekürzter Name des Wochentags
%A	voller Name des Wochentags
%b	abgekürzter Name des Monats
%B	voller Name des Monats
%c	Datum und Zeit
%d	Tag des Monats mit zwei Ziffern (01 bis 31)
%H	Stunde mit zwei Ziffern (00 bis 23)
%I	Stunde mit zwei Ziffern (01 bis 12)
%j	Tag des Jahres mit drei Ziffern (001 bis 366)
%m	Monat als Dezimalzahl mit zwei Ziffern
%M	Minute mit zwei Ziffern (00 bis 59)
%p	AM oder PM
%S	Sekunde mit zwei Ziffern (00 bis 59)
%U	Wochennummer mit zwei Ziffern, wobei Sonntag der erste Tag der Woche ist (00 bis 53)
%w	Wochentag, wobei Sonntag 0 ist (0 bis 6)
%W	Wochennummer mit zwei Ziffern, wobei Montag der erste Tag der Woche ist (00 bis 53)
%x	Datum
%X	Uhrzeit
%y	Jahr mit zwei Ziffern ohne Jahrhundertangabe (00 bis 99)
%Y	Jahr mit Jahrhundertangabe
%Z	Name der Zeitzone oder keine Zeichen, wenn keine Zeitzone festgelegt ist

Neben den in ANSI C definierten Formatierungsanweisungen werden auch die folgenden, von POSIX definierten Anweisungen unterstützt. Jede Formatierungsanweisung beginnt mit dem Prozentzeichen (%).

Um diese Formatierungsanweisungen verwenden zu können, müssen Sie __USELOCALES__ in Ihrem Programm definieren.

Formatierungsanweisung	steht für
%C	Jahrhundert als Dezimalzahl (00 ist 99)
%D	Datum im Format mm/dd/yy
%e	Tag des Monats als Dezimalzahl in einem zweistelligen Feld mit führenden Leerzeichen (1 bis 31)
%h	entspricht %b
%n	Zeilenvorschub
%r	Uhrzeit im 12-Stunden Format mit AM/PM, entspricht "%I:%M:%S %p"
%t	Tabulatorzeichen
%T	Uhrzeit im 24-Stunden Format "HH:MM:SS"
%u	Wochentag als Dezimalzahl (1 == Montag bis 7 == Sonntag)

Daneben unterstützt *strftime* die folgenden erweiterten Anweisungen, die ebenfalls von POSIX definiert sind:

Um diese Formatierungsanweisungen verwenden zu können, müsssen Sie __USELOCALES__ in Ihrem Programm definieren.

Formatierungsanweisung	steht für
%Od	Tag des Monats mit alternativen numerischen Symbolen
%Oe	Tag des Monats mit alternativen numerischen Symbolen
%OH	Stunde (24 Stunden) mit alternativen numerischen Symbolen
%OI	Stunde (12 Stunden) mit alternativen numerischen Symbolen
%Om	Monat mit alternativen numerischen Symbolen
%OM	Minuten mit alternativen numerischen Symbolen
%OS	Sekunden mit alternativen numerischen Symbolen
%Ou	Wochentag als Zahl mit alternativen numerischen Symbolen
%OU	Wochennummer mit alternativen numerischen Symbolen
%Ow	Wochentag als Zahl mit alternativen numerischen Symbolen
%OW	Wochennummer mit alternativen numerischen Symbolen
%Oy	Jahr mit alternativen numerischen Symbolen

Bei den mit %O modifizierten Formatierungsanweisungen wird der numerische Wert in die entsprechende Ordinalzahl konvertiert. Existiert die Ordinalzahl nicht, so wird die unmodifizierte Formatierungsanweisung verwendet.

Am 20.8.1988 würde die Formatierungsanweisung %d beispielsweise 20, die modifizierte Anweisung %Od aber 20^{th} liefern.

Rückgabewert

strftime liefert die Anzahl der Zeichen zurück, die nach *s* geschrieben wurden. Ist die Anzahl der benötigten Zeichen größer als *maxsize*, so wird 0 zurückgeliefert.

Siehe auch

localtime, *mktime*, *time*

stricmp, _fstricmp — string.h

Funktion

Vergleicht zwei Strings ohne Unterscheidung zwischen Groß- und Kleinschreibung.

Syntax

```
int stricmp(const char *s1, const char *s2);
int far _fstricmp(const char far *s1, const char far *s2);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

stricmp führt mit den Inhalten von *s1* und *s2* einen **unsigned**-Vergleich durch. Beginnend mit dem ersten Zeichen wird solange zeichenweise verglichen, bis die korrespondierenden Zeichen ungleich sind oder das Ende der Strings erreicht wird. Beim Vergleich werden Groß- und Kleinbuchstaben nicht unterschieden.

Zurückgeliefert wird ein Wert(<0, 0 oder >0) der auf dem Vergleich von *s1* (oder eines Teils davon) mit *s2* (oder eines Teils davon) basiert.

Die Routinen *stricmp* und *strcmpi* erfüllen dieselbe Funktion. *strcmpi* ist ein in string.h implementiertes Makro, das Aufrufe von *strcmpi* in Aufrufe von *stricmp* übersetzt. Daher müssen Sie die Header-Datei string.h einbinden, wenn Sie das Makro *strcmpi* verwenden wollen.

Rückgabewert *stricmp* liefert einen Wert

- < 0 wenn *s1* kleiner als *s2* ist
- == 0 wenn *s1* gleich *s2* ist
- > 0 wenn *s1* größer als *s2* ist

Siehe auch *_fstr**, *strcmp*, *strcmpi*, *strcoll*, *strncmp*, *strncmpi*, *strnicmp*

strlen, _fstrlen — string.h

Funktion Berechnet die Länge eines Strings.

Syntax

```
size_t strlen(const char *s);
size_t far _fstrlen(const char far *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strlen* berechnet die Länge des durch *s* angegebenen Strings.

Rückgabewert *strlen* liefert die Anzahl der Zeichen des Strings *s* zurück. Das abschließende Nullzeichen wird dabei nicht mitgezählt.

Siehe auch *_fstr**

strlwr, _fstrlwr — string.h

Funktion Konvertiert alle Großbuchstaben in einem String in Kleinbuchstaben.

Syntax

```
char *strlwr(char *s);
char far * far _fstrlwr(char far *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

S–T

Beschreibung *strlwr* konvertiert sämtliche Großbuchstaben des durch *s* übergebenen Strings in Kleinbuchstaben. Dabei wird die Kategorie LC_CTYPE der aktuellen locale-Einstellung berücksichtigt. In der locale-Einstellung »C« werden die Großbuchstaben *A* bis *Z* in die Kleinbuchstaben *a* bis *z* konvertiert. Andere Zeichen (insbesondere Umlaute) werden nicht berücksichtigt.

Rückgabewert *strlwr* liefert *s* zurück.

Siehe auch *_fstr**, *strupr*

strncat, _fstrncat string.h

Funktion Hängt einen Teilstring an einen anderen an.

Syntax

```
char *strncat(char *dest, const char *src, size_t maxlen);
char far *far _fstrncat(char far *dest, const char far *src, size_t maxlen);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strncat* hängt die ersten *maxlen* Zeichen von *src* an die bereits in *dest* vorhandenen Zeichen an. Die Operation wird beendet, wenn entweder maxlen Zeichen kopiert wurden oder das Ende von *src* erreicht ist. Als letztes schreibt *strncat* grundsätzlich ein Nullzeichen in *dest*. Die maximale Länge des resultierenden Strings beträgt *strlen*(*dest*) + *maxlen*.

Rückgabewert *strncat* liefert *dest* zurück.

Siehe auch *_fstr**

strncmp, _fstrncmp string.h

Funktion Vergleicht eine bestimmte Anzahl von Zeichen zweier Strings miteinander.

Syntax

```
int strncmp(const char *s1, const char *s2, size_t maxlen);
int far _fstrncmp(const char far *s1, const char far *s2, size_t maxlen);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strncmp* führt den gleichen **unsigned**-Vergleich wie *strcmp* durch, vergleicht aber maximal die ersten *maxlen* Zeichen von *s1* und *s2* miteinander. Beginnend mit dem ersten Zeichen wird zeichenweise verglichen, bis die korrespondierenden Zeichen ungleich sind oder *maxlen* Zeichen untersucht werden.

Rückgabewert *strncmp* vergleicht *s1* (oder einen Teil davon) mit *s2* (oder einen Teil davon) und liefert aufgrund des Vergleichs einen Integerwert

- < 0 wenn *s1* kleiner als *s2* ist
- == 0 wenn *s1* gleich *s2* ist
- > 0 wenn *s1* größer als *s2* ist.

Siehe auch *_fstr**, *strcmp*, *strcoll*, *stricmp*, *strncmpi*, *strnicmp*

strncmpi — string.h

Funktion

Vergleicht eine bestimmte Anzahl von Zeichen zweier Strings ohne Unterscheidung zwischen Groß- und Kleinschreibung.

Syntax

```
int strncmpi(const char *s1, const char *s2, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■				

Beschreibung

strncmpi vergleicht (in einem **signed**-Vergleich) maximal *n* Zeichen von *s1* und *s2* miteinander. Es wird kein Unterschied zwischen Groß- und Kleinschreibung gemacht. Beginnend mit dem ersten Zeichen wird zeichenweise solange verglichen, bis die korrespondierenden Zeichen ungleich oder *n* Zeichen untersucht worden sind. Zurückgeliefert wird ein Wert (<0, 0 oder >0), der auf dem Vergleich von *s1* (oder eines Teils davon) mit *s2* (oder eines Teils davon) basiert.

Die Routinen *strnicmp* und *strncmpi* erfüllen dieselbe Funktion; *strncmpi* ist ein in string.h implementiertes Makro, das Aufrufe von *strncmpi* in Aufrufe von *strnicmp* übersetzt. *strncmpi* wurde aus Kompatibilitätsgründen zu anderen Compilern definiert. Um *strncmpi* zu verwenden, muß daher die Header-Datei string.h eingebunden werden.

Rückgabewert

strncmpi liefert ein Wert

- < 0 wenn *s1* kleiner als *s2* ist
- == 0 wenn *s1* gleich *s2* ist
- > 0 wenn *s1* größer als *s2* ist

Siehe auch

strnicmp

strncpy, _fstrncpy — string.h

Funktion

Kopiert eine bestimmte Anzahl von Bytes von einem String in einen anderen, wobei gegebenenfalls verkürzt oder aufgefüllt wird.

Syntax

```
char *strncpy(char *dest, const char *src, size_t maxlen);
char far *far _fstrncpy(char far *dest, const char far *src, size_t maxlen);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

S–T

Beschreibung

strncpy kopiert bis zu *maxlen* Zeichen des über *src* angegebenen Strings in den durch *dest* angegebenen Speicherbereich. *strncpy* schreibt immer genau *maxlen* Zeichen – wenn *src* weniger als *maxlen* Zeichen enthält, wird eine entsprechende zusätzliche Anzahl von Nullzeichen in *dest* ange-

hängt. Wenn *src* mehr als *maxlen* Zeichen umfaßt, bleibt der nach *dest* kopierte String ohne ein abschließendes Nullzeichen.

Rückgabewert *strncpy* liefert *dest* zurück.

Siehe auch *_fstr**

strnicmp, _fstrnicmp — string.h

Funktion Vergleicht eine bestimmte Anzahl von Zeichen zweier Strings ohne Unterscheidung zwischen Groß- und Kleinschreibung.

Syntax

```
int strnicmp(const char *s1, const char *s2, size_t maxlen);
int far _fstrnicmp(const char far *s1, const char far *s2, size_t maxlen);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *strnicmp* vergleicht (in einem **signed**-Vergleich) maximal *maxlen* Zeichen von *s1* und *s2* miteinander. Es wird kein Unterschied zwischen Groß- und Kleinschreibung gemacht. Beginnend mit dem ersten Zeichen wird zeichenweise verglichen, bis die korrespondierenden Zeichen ungleich sind oder das Ende des Strings erreicht wird.

Zurückgeliefert wird ein Wert (<0, 0 oder >0), der auf dem Vergleich von *s1* (oder eines Teils davon) mit *s2* (oder eines Teils davon) basiert.

Rückgabewert *strnicmp* liefert einen Wert

- <0 wenn *s1* kleiner als *s2* ist
- ==0 wenn *s1* gleich *s2* ist
- >0 wenn *s1* größer als *s2* ist

Siehe auch *_fstr**

strnset, _fstrnset — string.h

Funktion Füllt eine angegebene Anzahl von Zeichen eines Strings mit einem bestimmten Zeichen.

Syntax

```
char *strnset(char *s, int ch, size_t n);
char far *far _fstrnset(char far *s, int ch, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *strnset* überschreibt die ersten *n* Zeichen des durch *s* angegebenen Strings mit dem Zeichen *ch*. Wenn *n* > *strlen*(*s*) ist, dann wird *n* durch *strlen*(*s*) ersetzt. *strnset* beendet die Operation, wenn *n* Zeichen überschrieben wurden oder wenn das abschließende Nullzeichen erreicht wird.

Rückgabewert *strnset* liefert *s* zurück.

Siehe auch *_fstr**

strpbrk, _fstrpbrk — string.h

Funktion Sucht einen String nach dem ersten Vorkommen bestimmter Zeichen ab.

Syntax

```
char *strpbrk(const char *s1, const char *s2);                    /* nur C */
char far *far _fstrpbrk(const char far *s1, const char far *s2)   /* C und C++ */

const char *strpbrk(const char *s1, const char *s2);               // nur C++
char *strpbrk(char *s1, const char *s2);                           // nur C++
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strpbrk* sucht den über *s1* angegebenen String nach dem ersten Vorkommen eines in *s2* enthaltenen Zeichens ab.

Rückgabewert Die Funktion liefert einen Zeiger auf die erste Fundstelle eines Zeichens aus *s2* zurück. Erscheint kein Zeichen aus *s2* in *s1*, dann liefert *strpbrk* den Wert NULL zurück.

Siehe auch *_fstr**

strrchr, _fstrrchr — string.h

Funktion Sucht einen String nach dem letzten Vorkommen eines bestimmten Zeichens ab.

Syntax

```
char *strrchr(const char *s, int c);                     /* nur C */
char far *far _fstrrchr(const char far *s, int c);      /* C und C++ */

const char *strrchr(const char *s, int c);                // nur C++
char *strrchr(char *s, int c);                            // nur C++
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strrchr* sucht den über *s* angegebenen String nach dem Zeichen *c* ab, wobei die Suche mit dem *letzten* Zeichen von *s* beginnt. Das abschließende Nullzeichen wird zum Stringinhalt gerechnet und als erstes verglichen.

Rückgabewert *strrchr* liefert einen Zeiger auf das letzte Vorkommen des Zeichens *c* in *s* zurück bzw. den Wert NULL, wenn der String dieses Zeichen nicht enthält.

Siehe auch *_fstr*, strcspn, strchr*

strrev, _fstrrev — string.h

Funktion Dreht die Reihenfolge der Zeichen eines Strings um.

Syntax

```
char *strrev(char *s);
char far *far _fstrrev(char far *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *strrev* spiegelt die Reihenfolge sämtlicher Zeichen des über *s* angegebenen Strings (aus »Meier« wird »reieM«). Das abschließende Nullzeichen bleibt unverändert.

Rückgabewert *strrev* liefert einen Zeiger auf den gespiegelten String zurück.

Siehe auch *_fstr**

strset, _fstrset — string.h

Funktion Überschreibt sämtliche Zeichen eines Strings mit einem bestimmten Zeichen.

Syntax

```
char *strset(char *s, int ch);
char far *far _fstrset(char far *s, int ch);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *strset* überschreibt sämtliche Zeichen des durch *s* angegebenen Strings mit dem Zeichen *ch* und bricht erst beim abschließenden Nullzeichen ab.

Rückgabewert *strset* liefert *s* zurück.

Siehe auch *_fstr*, setmem*

strspn, _fstrspn — string.h

Funktion Liefert die Länge des Teilstrings von *s1* zurück, der ausschließlich aus den in *s2* angegebenen Zeichen besteht.

Syntax

```
size_t strspn(const char *s1, const char *s2);
size_t far _fstrspn(const char far *s1, const char far *s2);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Beginnend mit dem ersten Zeichen überprüft die Funktion *strspn*, ob die Zeichen in *s1* Bestandteil von *s2* sind. Die Funktion stoppt beim ersten Zeichen von *s1*, das nicht in *s2* enthalten ist.

Rückgabewert *strspn* liefert die Länge des Teilstrings von *s1*, der ausschließlich aus Zeichen besteht, die in *s2* enthalten sind.

Siehe auch *_fstr**

strstr, _fstrstr — string.h

Funktion Sucht einen String nach dem Vorkommen eines bestimmten Teilstrings ab.

Syntax

```
char *strstr(const char *s1, const char *s2);                     /* nur C */
char far * far _fstrstr(const char far *s1, const char far *s2);  /* C und C++ */

const char *strstr(const char *s1, const char *s2);                // nur C++
char *strstr(char *s1, const char *s2);                            // nur C++
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strstr* sucht den über *s1* angegebenen String nach dem ersten Vorkommen des Teilstrings *s2* ab.

Rückgabewert *strstr* liefert einen Zeiger auf den Beginn von *s2* in *s1* zurück bzw. den Zeigerwert NULL, wenn *s2* in *s1* überhaupt nicht vorkommt.

Siehe auch *_fstr**

S–T

_strtime — time.h

Funktion Konvertiert die aktuelle Uhrzeit in einen String.

Syntax

```
char *_strtime(char *buf);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

_strtime konvertiert die aktuelle Uhrzeit in einen String und speichert diesen in dem durch *buf* angegebenen Puffer. *buf* muß mindestens 9 Zeichen lang sein.

Der String ist durch ein Nullzeichen abgeschlossen und hat das folgende Format

```
HH:MM:SS.
```

HH, MM und SS sind jeweils Zahlen mit zwei Ziffern und geben Stunde, Minute und Sekunde an. Der String ist nullterminiert.

Rückgabewert

Der Rückgabewert von *_strtime* ist *buf*, die Adresse des Uhrzeit-Strings.

Siehe auch

asctime, *ctime*, *localtime*, *strftime*, *_strdate*, *time*

strtod, _strtold — stdlib.h

Funktion

Konvertiert einen String in einen **double**- bzw. **long double**-Wert.

Syntax

```
double strtod(const char *s, char **endptr);
long double _strtold(const char *s, char **endptr);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
strtod	■	■	■	■	■	■	■
_strtold	■		■	■			■

Beschreibung

strtod konvertiert den durch *s* angegebenen String in einen **double**-Wert. Der Stringinhalt muß daher als **double** interpretierbar sein und das folgende Format haben:

```
[ws] [sn] [ddd] [.] [ddd] [fmt [sn] ddd]
```

mit

[ws] = optionales Whitespace-Zeichen
[sn] = für ein optionales Vorzeichen (+ oder -)
[ddd] = optionale Ziffern
[fmt] = optionales Exponentzeichen (*e* oder *E*)
[.] = optionaler Dezimalpunkt

strtod erkennt außerdem +INF und -INF für plus bzw. minus unendlich, sowie +NAN und -NAN für Not-A-Number (keine darstellbare Zahl).

Beispiele für gültige Angaben sind:

+1234.1981e-1
502.85E2
+2010.952

strtod bricht die Analyse beim ersten Zeichen ab, das nicht mehr als Teil eines **double**-Werts interpretiert werden kann.

Solange der Parameter *endptr* nicht NULL ist, wird **endptr* von *strtod* auf das Zeichen innerhalb von *s* gesetzt, durch das die Analyse abgebrochen wurde (**endptr = &abbruch_zeichen*). *endptr* ist für die Fehlersuche nützlich.

_strtold ist die **long double** Version. Sie konvertiert einen String in einen Wert vom Typ **long double**.

Rückgabewert Diese Funktionen liefern das Ergebnis der Konvertierung von *s* als **double** (*strtod*) oder als **long double** (*_strtold*) zurück. Im Falle eines Überlaufs wird plus oder minus HUGE_VAL (*strtod*) bzw. _LHUGE_VAL (*_strtold*) zurückgeliefert.

Siehe auch *atof*

strtok, _fstrok — string.h

Funktion Sucht einen String nach einer von mehreren Zeichenfolgen ab, die in einem zweiten String definiert sind.

Syntax

```
char *strtok(char *s1, const char *s2);
char far *far _fstrtok(char far *s1, const char far *s2);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *strtok* betrachtet den über *s1* angegebenen String als Folge von null oder mehr Zeichenfolgen (Token), die durch eines oder mehrere der in *s2* definierten Zeichen voneinander getrennt sind.

Der erste Aufruf von *strtok* liefert einen Zeiger auf den Beginn der ersten gefundenen Zeichenfolge in *s1* zurück und setzt ein Nullzeichen unmittelbar hinter das Ende der Zeichenfolge. Folgende Aufrufe von *strtok* mit dem Wert NULL anstelle des Arguments *s1* liefern die restlichen Zeichenfolgen in *s1*.

s2 kann bei jedem Aufruf von *strtok* beliebig geändert werden.

Rückgabewert *strtok* liefert einen Zeiger auf den Beginn der gefundenen Zeichenfolge in *s1* zurück. Wenn keine weitere Zeichenfolge gefunden wird, ist der Rückgabewert NULL.

Siehe auch *_fstr**

strtol stdlib.h

Funktion Konvertiert einen String in einen **long**-Wert.

Syntax `long strtol(const char *s, char **endptr, int radix)`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *strtol* konvertiert den durch *s* angegebenen String in einen **long**-Wert. Der Stringinhalt muß daher als **long**-Wert interpretierbar sein und das folgende Format haben:

```
[ws] [sn] [0] [x] [ddd]
```

mit

[ws] = optionales Whitespace-Zeichen
[sn] = optionales Vorzeichen (+ oder -)
[0] = eine optionale Null (0)
[x] = ist ein optionales x oder X
[ddd] = optionale Ziffern

strtol bricht den Lesevorgang ab, wenn ein Zeichen nicht dem obigen Format entspricht.

Liegt *radix* zwischen 2 und 36, so wird *s* im Zahlensystem zur Basis *radix* interpretiert. Ist *radix* 0, so legen die ersten Zeichen von *s* die verwendete Basis fest:

Erstes Zeichen	Zweites Zeichen	Interpretation
0	1 bis 7	Oktalsystem
0	x oder X	Hexadezimalsystem
1 bis 9		Dezimalsystem

Ist *radix* gleich 1, kleiner als 0 oder größer als 36, so wird der Parameter als ungültiger Wert interpretiert.

Bei Ausgabe ungültiger Werte für *radix* liefert die Funktion null zurück und setzt den Zeiger **endptr* auf den Stringanfang.

Falls der Wert in *s* als Oktalzahl interpretiert werden soll, werden nur die Zeichen zwischen 0 und 7 erkannt.

Soll der Wert in *s* als Dezimalzahl interpretiert werden, werden nur die Zeichen 0 bis 9 erkannt.

Falls der Wert in *s* als Zahl eines Zahlensystems zu einer anderen Basis interpretiert werden soll, werden nur die Zeichen, die zur Repräsentation von Zahlen dieses Zahlensystems verwendet werden, erkannt. (Beispielsweise nur die Zeichen 0 bis 4, wenn *radix* gleich 5 ist, ist *radix* gleich 20, werden nur die Zeichen 0 bis 9 und A bis J erkannt.)

Wenn *endptr* ungleich 0 ist, dann wird **endptr* auf die Adresse des ersten nicht analysierbaren Zeichens in *s* gesetzt (**endptr* = *&abbruch_zeichen*).

Rückgabewert *strtol* liefert das Ergebnis der Konvertierung als **long**-Wert, im Fehlerfall den Wert 0 zurück.

Siehe auch *atoi*, *atol*, *strtoul*

_strtold

Siehe *strtod*.

strtoul stdlib.h

Funktion Konvertiert einen String in einen **unsigned long**-Wert des durch *radix* gegebenen Zahlensystems.

Syntax

```
unsigned long strtoul(const char *s, char **endptr, int radix);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung *strtoul* arbeitet genauso wie *strtol*, konvertiert die durch *s* angegebene Zeichenfolge aber in einen **unsigned long**-Wert (siehe die Beschreibung von *strtol*).

Rückgabewert *strtoul* liefert das Ergebnis der Konvertierung als **unsigned long**-Wert, im Fehlerfall den Wert 0 zurück.

Siehe auch *atol*, *strtol*

strupr, _fstrupr string.h

Funktion Konvertiert alle Kleinbuchstaben eines Strings in Großbuchstaben.

Syntax

```
char *strupr(char *s);
char far *far _fstrupr(char far *s);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

strupr konvertiert die Kleinbuchstaben des durch *s* angegebenen Strings in Großbuchstaben. Dabei wird die Kategorie LC_TYPE der aktuellen locale-Einstellung berücksichtigt. In der standardmäßigen locale-Einstellung »C« werden die Buchstaben *a* bis *z* in die Buchstaben *A* bis *Z* konvertiert und alle anderen Zeichen (insbesondere die deutschen Umlaute) bleiben unverändert.

Rückgabewert

strupr liefert *s* zurück.

Siehe auch

_fstr, strlwr*

strxfrm — string.h

Funktion

Transformiert einen String anhand einer bestimmten Sortierreihenfolge.

Syntax

```
size_t strxfrm(char *target, constchar *source, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■	■	■	■

Beschreibung

strxfrm transformiert maximal *n* Zeichen des durch *source* angegebenen Strings so in den String *target*, daß die Funktion *strcmp*, angewendet auf die Ergebnisstrings der Funktion, dieselben Werte zurückliefert wie die Funktion *strcoll*, angewendet auf die Ausgangsstrings.

Einschließlich des abschließenden Nullzeichens werden maximal *n* Zeichen von *source* transformiert.

Der Ergebnisstring richtet sich nach der Kategorie LC_COLLATE der aktuellen locale-Einstellung. Der generierte Spezialstring kann mit anderen Strings desselben Typs byteweise (beispielsweise per *strcmp*) verglichen werden. Die Vergleichsergebnisse entsprechen dann der aktuellen locale-Einstellung für Vergleiche. Diese Spezialstrings können als Schlüssel oder tokenisierte Strings aufgefaßt werden, und sind jeweils nur innerhalb einer locale-Einstellung sinnvoll.

Die in *target* enthaltenen Token werden anhand der von *strcoll* verwendeten Vergleichsgewichtungen aus den Vergleichstabellen der aktuellen locale-Einstellung erzeugt.

Die Funktion wird beendet, wenn alle Ebenen bearbeitet wurden oder wenn die Länge des tokenisierten Strings den Parameter *n* erreicht.

Alle überflüssigen Token werden aus der Tokenmenge jeder Ebene entfernt.

Der Puffer für den tokenisierten String muß groß genug sein. Seine Größe hängt von der Länge des Strings *source*, von der Anzahl der Vergleichsebenen, von den Regeln für jede Ebene sowie vom Auftreten von Sonderzeichen ab. Bestimmte Sonderzeichen verursachen zusätzliche Bearbeitungsschritte und damit erhöhten Platzbedarf. Das französische Zeichen »oe« benötigt beispielsweise den doppelten Platz, da es in einigen locale-Einstellungen pro Ebene auf zwei Vergleichsgewichtungen erweitert wird. Teilstrings mit Ersetzungen benötigen ebenfalls zusätzlichen Platz.

Für die Berechnung des benötigten Speicherplatzes gibt es keine allgemeingültige Formel. Es sind jedoch mindestens (Ebenen * Stringlänge) Bytes erforderlich.

Rückgabewert *strxfrm* liefert die Anzahl der transformierten Zeichen zurück. Das abschließende Nullzeichen wird nicht mitgezählt. Ist der zurückgelieferte Wert gleich oder größer als *n*, dann ist der Inhalt von *target* unbestimmt.

Siehe auch *strcmp*, *strcoll*, *strncpy*

swab — stdlib.h

Funktion Vertauscht Bytepositionen.

Syntax

```
void swab(char *from, char *to, int nbytes);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *swab* liest *nbytes* Bytes von dem durch *from* bezeichneten Speicherbereich, tauscht jeweils die Worthälften miteinander aus (das erste Byte wird mit dem zweiten vertauscht, das dritte mit dem vierten usw.), und schreibt diese vertauschten Bytes in den durch *to* bezeichneten Speicherbereich. Der Parameter *nbytes* sollte einen geradzahligen Wert haben. Sinnvoll ist *swab* hauptsächlich dann, wenn Daten zwischen Maschinen mit Prozessoren ausgetauscht werden sollen, die mit unterschiedlichen Byte-Reihenfolgen arbeiten.

Rückgabewert Keiner.

system — stdlib.h

Funktion Führt einen Betriebssystembefehl aus.

Syntax

```
int system(const char *command);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■			■

Beschreibung

system ruft innerhalb eines ausgeführten C-Programms den Kommandozeileninterpreter des Betriebssystems zur Ausführung eines Befehls, einer Batch-Datei oder eines Programms auf. Dazu wird ihm der String *command* als Kommandozeile übergeben.
Aufgerufene Programme müssen sich im aktuellen Verzeichnis bzw. in einem in der Umgebungsvariable PATH aufgeführten Verzeichnis befinden.

Der Kommandozeileninterpreter wird mit Hilfe der Umgebungsvariable COMSPEC gefunden, so daß sich die Datei des Kommandozeileninterpreters nicht im aktuellen Verzeichnis befinden muß.

Rückgabewert

system liefert einen Wert ungleich null, falls *command* ein NULL-Zeiger ist und der Kommandozeileninterpreter zur Verfügung steht.

Ist *command* kein NULL-Zeiger, so liefert *system* den Wert 0 zurück, falls der Kommandozeileninterpreter erfolgreich gestartet wurde.

Im Fehlerfall wird der Wert -1 zurückgeliefert und *errno* auf einen der folgenden Werte gesetzt:

ENOENT	Pfad/Dateiname nicht gefunden
ENOEXEC	*exec*-Formatfehler
ENOMEM	Nicht genug Platz im Hauptspeicher

Siehe auch

exec..., _fpreset, searchpath, spawn...

tan, tanl — math.h

Funktion

Liefert den Tangens des Arguments.

Syntax

```
double tan(double x);
long double tanl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
tan	■	■	■	■	■	■	■
tanl	■		■	■			■

Beschreibung

tan erwartet einen Winkel in der Einheit *rad* als Argument und liefert den dazugehörigen Tangens *sin(x)/cos(x)* zurück.

tanl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück.

Die Fehlerbehandlung für *tan* und *tanl* kann mit *_matherr* bzw. *_matherrl* modifiziert werden.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert *tan* und *tanl* liefern den Tangens von *x* zurück.

Siehe auch *acos, asin, atan, atan2, bcd, complex, cos, sin*

tanh, tanhl — math.h

Funktion Liefert den Tangens hyperbolicus des Arguments.

Syntax

```
double tanh(double x);
long double tanhl(long double x);
```

	DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
tanh	■	■	■	■	■	■	■
tanhl	■		■	■			■

Beschreibung *tanh* berechnet den Tangens hyperbolicus (*sinh(x)* / *cosh(x)*).

tanhl ist die **long double** Version. Sie erwartet ein Argument vom Typ **long double** und liefert ein Ergebnis vom Typ **long double** zurück. Die Fehlerbehandlung für *tanh* und *tanhl* kann mit *_matherr* bzw. *_matherrl* modifiziert werden.

Diese Funktion kann mit den Datentypen *bcd* und *complex* verwendet werden.

Rückgabewert *tanh* und *tanhl* liefern den Tangens hyperbolicus von *x*.

Siehe auch *bcd, complex, cos, cosh, sin, sinh, tan*

tell — io.h

Funktion Liefert die aktuelle Position eines Dateizeigers zurück.

Syntax

```
long tell(int handle);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

S–T

Beschreibung *tell* liefert die aktuelle Position des Dateizeigers innerhalb der durch *handle* bezeichneten Datei als Byte-Zahl relativ zum Dateianfang zurück.

Rückgabewert *tell* liefert die aktuelle Position des Dateizeigers innerhalb der angegebenen Datei als **long**-Wert zurück. Im Fehlerfall wird *-1* zurückgeliefert und die globale Variable *errno* erhält den Wert:

EBADF	Ungültiges Handle

Siehe auch *fgetpos*, *fseek*, *ftell*, *lseek*

tempnam stdio.h

Funktion Erzeugt einen eindeutigen Dateinamen im angegebenen Verzeichnis.

Syntax

```
char *tempnam(char *dir, char *prefix);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung Die Funktion *tempnam* erzeugt eindeutige Dateinamen in beliebigen Verzeichnissen. Die zugehörige Datei wird dabei nicht erzeugt – *tempnam* gewährleistet lediglich, daß eine Datei mit diesem Namen noch nicht existiert. *tempnam* versucht in den folgenden Verzeichnissen (in der angegebenen Reihenfolge) einen eindeutigen Dateinamen zu erzeugen:

- In dem durch die Umgebungsvariable TMP angegebenen Verzeichnis.
- In dem Verzeichnis, das mit *dir* an *tempnam* übergeben wird.
- Im Verzeichnis, das in der *P_tmpdir*-Definition in stdio.h angegeben ist. Wenn Sie dieses Verzeichnis in stdio.h ändern, wird die neue Definition *nicht* von *tempnam* verwendet.
- Im aktuellen Arbeitsverzeichnis.

Wenn eines der aufgeführten Verzeichnisse NULL, nicht definiert oder nicht vorhanden ist, wird es ignoriert.

Der Parameter *prefix* enthält den ersten Teil des Dateinamens. Dieser darf nicht länger als fünf Zeichen sein und keinen Punkt (.) enthalten. Durch die Verknüpfung des Verzeichnisnamens mit dem Parameter *prefix* und 6 zusätzlichen Zeichen wird ein eindeutiger Dateiname erzeugt. Der Speicherplatz für den resultierenden Dateinamen wird mit *malloc* reserviert. Wenn der Dateiname nicht mehr benötigt wird, sollte die aufrufende Funktion den Speicherplatz durch einen Aufruf von *free* wieder freigeben.

Im Gegensatz zur Funktion *tmpfile* löscht *tempnam* den Datei nicht. Wenn Sie mit dem über *tempnam* erzeugten Namen eine temporäre Datei erstellen, liegt es deshalb in Ihrer Verantwortung, die Datei vor dem Ende des Programms wieder zu löschen (z.B. mit *remove*).

Rückgabewert Bei fehlerfreier Ausführung liefert *tempnam* einen Zeiger auf den eindeutigen Namen einer temporären Datei. Dieser Zeiger kann später (wenn der Dateiname nicht mehr benötigt wird) an *free* übergeben werden. Wenn *tempnam* keinen eindeutigen Dateinamen zurückliefern kann, ist das Funktionsergebnis NULL.

Siehe auch *mktemp*, *tmpfile*, *tmpnam*

textattr — conio.h

Funktion Setzt das Attribut für folgende Textausgaben.

Syntax

```
void textattr(int newattr);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *textattr* setzt sowohl die Vordergrund- als auch die Hintergrundfarbe für folgende Textausgaben (d.h. vereinigt sozusagen die Funktionen *textcolor* und *textbackground*).

Diese Funktion läßt bereits auf dem Bildschirm befindliche Zeichen unverändert. Sie verändert nur die Zeichen, die im Textmodus von Funktionen (wie *cprintf*) direkt auf den Bildschirm ausgegeben werden, nachdem die Funktion aufgerufen wurde.

Der Parameter *newattr* hat die folgende Struktur:

7	6	5	4	3	2	1	0
B	b	b	b	f	f	f	f

Der 8 Bit Parameter *newattr* ist folgendermaßen aufgebaut:

- *ffff* ist die 4 Bit Vordergrundfarbe (0 bis 15).
- *bbb* ist die 3 Bit Hintergrundfarbe (0 bis 7).
- *B* steuert das Attribut »Blinken«.

Bei gesetztem Blink-Bit blinkt das Zeichen.

Wenn die in conio.h definierten Farbkonstanten zur Zuordnung von Textattributen über *textattr* verwendet werden sollen, müssen folgende Beschränkungen beachtet werden:

- Sie können nur eine der acht ersten Farben für den Hintergrund wählen.

- Die Hintergrundfarbe muß um 4 Bitpositionen nach links geschoben werden.

Für Vorder- und Hintergrundfarben sind die folgenden symbolischen Konstanten in conio.h definiert:

Konstante	Wert	Beschreibung	Vorder- oder Hintergrund?
BLACK	0	schwarz	Für beide
BLUE	1	blau	Für beide
GREEN	2	grün	Für beide
CYAN	3	türkis	Für beide
RED	4	rot	Für beide
MAGENTA	5	violett	Für beide
BROWN	6	braun	Für beide
LIGHTGRAY	7	hellgrau	Für beide
DARKGRAY	8	dunkelgrau	Nur für Vordergrund
LIGHTBLUE	9	hellblau	Nur für Vordergrund
LIGHTGREEN	10	hellgrün	Nur für Vordergrund
LIGHTCYAN	11	helltürkis	Nur für Vordergrund
LIGHTRED	12	hellrot	Nur für Vordergrund
LIGHTMAGENTA	13	hellviolett	Nur für Vordergrund
YELLOW	14	gelb	Nur für Vordergrund
WHITE	15	weiß	Nur für Vordergrund
BLINK	128	»blinkend«	Nur für Vordergrund

➡ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *gettextinfo, highvideo, lowvideo, normvideo, textbackground, textcolor*

textbackground — conio.h

Funktion Legt die Hintergrundfarbe für nachfolgende Textausgaben fest.

Syntax

```
void textbackground(int newcolor);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *textbackground* legt die Hintergrundfarbe für folgende Zeichenausgaben fest, die im Textmodus direkt auf den Bildschirm geschrieben werden. Der Parameter *newcolor* legt die neue Hintergrundfarbe fest. Er kann auf einen Integerwert zwischen 0 und 7 oder auf eine der in conio.h definierten Konstanten gesetzt werden. Falls Sie symbolische Konstanten verwenden, muß die Header-Datei conio.h eingebunden werden.

Wenn *textbackground* aufgerufen wurde, verwenden alle folgenden Funktionsaufrufe mit direkter Bildschirmausgabe (wie z.B. *cprintf*) den Parameter *newcolor*. Bereits auf dem Bildschirm befindliche Zeichen bleiben unverändert.

Die Werte für die Hintergrundfarbe entsprechen folgenden in conio.h definierten symbolischen Konstanten:

Name	Berschreibung	Wert
BLACK	schwarz	0
BLUE	blau	1
GREEN	grün	2
CYAN	türkis	3
RED	rot	4
MAGENTA	violett	5
BROWN	braun	6
LIGHTGRAY	hellgrau	7

➥ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *gettextinfo*, *textattr*, *textcolor*

textcolor conio.h

Funktion Legt die Zeichenfarbe für folgende Textausgaben fest.

Syntax

```
void textcolor(int newcolor);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *textcolor* legt die Zeichenfarbe für folgende Ausgaben fest. Der zulässige Wertebereich für *newcolor* ist 0 bis 15 oder einer der in conio.h definierten symbolischen Konstanten. Falls Sie symbolische Konstanten verwenden, muß die Header-Datei conio.h eingebunden werden.

Wenn *textcolor* aufgerufen wurde, verwenden alle folgenden Aufrufe von Funktionen mit direkter Bildschirmausgabe (wie z.B. *cprintf*) den Parameter *newcolor*. Bereits auf dem Bildschirm befindliche Zeichen bleiben unverändert.

Die folgenden symbolischen Konstanten für die Zeichenfarbe sind in conio.h definiert:

Name	Beschreibung	Wert
BLACK	schwarz	0
BLUE	blau	1
GREEN	grün	2
CYAN	türkis	3
RED	rot	4
MAGENTA	violett	5
BROWN	braun	6
LIGHTGRAY	hellgrau	7
DARKGRAY	dunkelgrau	8
LIGHTBLUE	hellblau	9
LIGHTGREEN	hellgrün	10
LIGHTCYAN	helltürkis	11
LIGHTRED	hellrot	12
LIGHTMAGENTA	hellviolett	13
YELLOW	gelb	14
WHITE	weiß	15
BLINK	»blinkend«	128

Durch Addition des Wertes 128 zur Vordergrundfarbe können Sie Zeichen blinkend darstellen lassen. Zu diesem Zweck wurde auch die Konstante BLINK definiert. Sie verwenden sie wie folgt:

```
textcolor (CYAN + BLINK);
```

Nicht alle Farbmonitore reagieren auf das Intensitätsbit, das die Farben 0 bis 7 und 8 bis 15 voneinander unterscheidet. Wie die verschiedenen Farben auf Systemen mit monochromen, LCD- und Plasmabildschirmen dargestellt werden (verschiedene Helligkeitsgrade, spezielle Muster oder Attribute wie unterstrichen, fett, etc.), hängt von der jeweiligen Hardware ab.

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *gettextinfo*, *highvideo*, *lowvideo*, *normvideo*, *textattr*, *textbackground*

textmode — conio.h

Funktion Setzt einen Textmodus.

Syntax `void textmode(int newmode);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

textmode setzt einen bestimmten Textmodus.

Sie können den Textmodus über den Parameter *textmode* mit Hilfe des in conio.h definierten Aufzählungstyps *text_modes* setzen.

Die am häufigsten benutzten *text_modes*-Werte werden in der folgenden Tabelle aufgeführt. In conio.h finden Sie einige weitere Werte.

Name	Beschreibung
LASTMODE	vorheriger Textmodus
BW40	schwarz-weiß, 40 Spalten
C40	farbig, 40 Spalten
BW80	schwarz-weiß, 80 Spalten
C80	farbig, 80 Spalten
MONO	schwarz-weiß, 80 Spalten für Monochrom-Adapter bzw. Hercules-Karte
C4350	EGA 43-Zeilen und VGA 50-Zeilen-Modus

Beim Aufruf von *textmode* wird das aktuelle Fenster auf die gesamte Bildschirmgröße vergrößert, und die Textattribute werden wie beim Aufruf von *normvideo* auf ihre Standardwerte zurückgesetzt.

Ein Aufruf von *textmode* mit der Konstanten LASTMODE als Parameter ist nur dann sinnvoll, wenn der Bildschirm bereits im Textmodus ist. Der Bildschirm wird dann auf den zuletzt verwendeten Textmodus umgeschaltet.

textmode sollte generell nur verwendet werden, wenn der Bildschirm bereits im Textmodus ist. Befindet sich der Bildschirm im Grafikmodus, so sollten Sie stattdessen *restorecrtmode* zum vorübergehenden Umschalten in den Textmodus verwenden.

➥ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *gettextinfo*, *window*

time — time.h

Funktion Liefert Datum und Uhrzeit des Systems.

Syntax `time_t time(time_t *timer);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *time* setzt die mit *timer* übergebene Variable auf die Anzahl der Sekunden, die seit dem 1. Januar 1970, 00:00:00 (Greenwich Mean Time) verstrichen sind, sofern *timer* kein NULL-Zeiger ist.

Rückgabewert *time* liefert die Anzahl der seit dem 1.1.1970 verstrichenen Sekunden zurück.

Siehe auch *asctime*, *ctime*, *difftime*, *ftime*, *gettime*, *gmtime*, *localtime*, *settime*, *stime*, *tzset*

tmpfile stdio.h

Funktion Öffnet eine temporäre Datei im Binärmodus.

Syntax `FILE *tmpfile(void);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *tmpfile* erzeugt eine temporäre binäre Datei und öffnet diese Datei für Lese- und Schreibzugriffe (w+b). Diese Datei wird beim Schließen bzw. bei Programmende automatisch wieder gelöscht.

Rückgabewert Bei fehlerfreier Ausführung liefert *tmpfile* einen Zeiger auf den Stream der erzeugten Temporärdatei zurück. Wenn die Datei aus irgendeinem Grund nicht erzeugt oder geöffnet werden konnte, ist das Funktionsergebnis NULL.

Siehe auch *fopen*, *tmpnam*

tmpnam stdio.h

Funktion Erzeugt eindeutige Namen für temporäre Dateien.

Syntax `char *tmpnam(char *s);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *tmpnam* erzeugt bei jedem Aufruf einen neuen String und kann für die Benennung temporärer Dateien verwendet werden. Die Funktion erzeugt

bis zu TMP_MAX verschiedene Dateinamen. TMP_MAX ist in stdio.h mit dem Wert 65.535 definiert.

Wenn der als *s* übergebene Zeiger den Wert NULL hat, liefert *tmpnam* einen Zeiger auf einen statischen Puffer zurück, der durch jeden Aufruf dieser Funktion erneut gesetzt wird. Wenn *s* einen Wert ungleich NULL hat, erwartet *tmpnam*, daß dieser Zeiger auf ein **char**-Array mit mindestens *L_tmpnam* Elementen zeigt, und kopiert das Funktionsergebnis in dieses Array. Die Konstante *L_tmpnam* ist in stdio.h definiert.

➥ Wenn Sie mit *tmpname* eine Temporärdatei erzeugen, so liegt das Löschen der Datei (beispielsweise durch einen Aufruf von *remove*) in Ihrer Verantwortung. Die Datei wird nicht (wie etwa bei *tmpfile*) automatisch gelöscht.

Rückgabewert Wenn *s* den Wert NULL hat, liefert *tmpnam* einen Zeiger auf einen internen statischen Puffer, ansonsten den Zeiger *s* zurück.

Siehe auch *tmpfile*

toascii — ctype.h

Funktion Konvertiert Zeichen in das ASCII-Format.

Syntax

```
int toascii(int c);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *toascii* ist ein Makro, das den übergebenen Wert *c* (durch Löschen aller Bits, außer der 7 niederwertigen) auf den Bereich 0 bis 127 begrenzt und das Ergebnis dieser Operation zurückliefert.

Rückgabewert Das Makro liefert das Ergebnis der Konvertierung von *c* als Wert vom Typ **int** zurück.

_tolower — ctype.h

Funktion Wandelt Groß- in Kleinbuchstaben um.

Syntax

```
int _tolower(int ch);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *_tolower* ist ein Makro, das wie die Funktion *tolower* das übergebene Zeichen *ch* in einen Kleinbuchstaben umwandelt. Dieses Makro ist jedoch

nur auf Großbuchstaben (A bis Z) anwendbar – für Ziffern und Sonderzeichen erzeugt es unsinnige Ergebnisse.

_tolower kann nur verwendet werden, wenn die Header-Datei ctype.h eingebunden ist.

Rückgabewert Das Makro liefert als Ergebnis der Konvertierung einen Wert vom Typ **int** zurück, falls *ch* ein Großbuchstabe ist; ansonsten ist das Resultat undefiniert.

tolower ctype.h

Funktion Wandelt Groß- in Kleinbuchstaben um.

Syntax

```
int tolower(int ch);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *tolower* ist eine Funktion, die das als Integerwert übergebene Zeichen *ch* (ein Wert im Bereich von EOF bis 255) in einen Kleinbuchstaben umwandelt. Die Funktion ist von der Kategorie LC_TYPE der aktuellen locale-Einstellung abhängig. In der Standardeinstellung »C« wird *ch* in einen Kleinbuchstaben (*a* bis *z*) umgewandelt, sofern *ch* einen Großbuchstaben (*A* bis Z) enthält. Alle anderen Werte werden nicht verändert.

Rückgabewert *tolower* liefert den konvertierten Wert von *ch* zurück, falls *ch* einen Großbuchstaben enthält. Ansonsten wird das Zeichen unverändert zurückgeliefert.

_toupper ctype.h

Funktion Wandelt Klein- in Großbuchstaben um.

Syntax

```
int _toupper(int ch);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *_toupper* ist ein Makro und wandelt wie die Funktion *toupper* Kleinbuchstaben in Großbuchstaben um. Im Unterschied zur Funktion muß beim Makro der Parameter *ch* einen Kleinbuchstaben (*a* bis *z*) enthalten.

_toupper kann nur verwendet werden, wenn die Header-Datei ctype.h eingebunden ist.

Rückgabewert *_toupper* liefert den konvertierten Wert von *ch* zurück, falls *ch* einen Großbuchstaben enthält. Ansonsten ist das Ergebnis undefiniert.

toupper ctype.h

Funktion Wandelt Klein- in Großbuchstaben um.

Syntax

```
int toupper(int ch);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *toupper* ist eine Funktion, die das als Integerwert übergebene Zeichen *ch* (ein Wert im Bereich von EOF bis 255) in einen Großbuchstaben umwandelt. Die Funktion ist von der Kategorie LC_TYPE der aktuellen locale-Einstellung abhängig. In der Standardeinstellung »C« wird *ch* in einen Großbuchstaben (*A* bis *Z*) umgewandelt, sofern *ch* einen Kleinbuchstaben (*a* bis *z*) enthält. Alle anderen Werte werden nicht verändert.

Rückgabewert *toupper* liefert den konvertierten Wert von *ch* zurück, falls *ch* einen Kleinbuchstaben enthält. Ansonsten wird das Zeichen unverändert zurückgeliefert.

tzset time.h

Funktion Setzt die globalen Variablen *_daylight*, *_timezone* und *_tzname*.

Syntax

```
void tzset(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *tzset* ist auf XENIX-Systemen verfügbar.

Die Funktion setzt die globalen Variablen *_daylight*, *_timezone* und *_tzname* gemäß dem Inhalt der Umgebungsvariable TZ. Die Bibliotheksfunktionen *ftime* und *localtime* verwenden diese globalen Variablen, um die Greenwich Mean Time (GMT) an die lokale Zeitzone anzupassen. Die Umgebungsvariable TZ muß folgendes Format besitzen:

```
TZ = zzz[+|-]d[d][lll]
```

Das Feld *zzz* steht für den Namen der Zeitzone, der aus drei Buchstaben bestehen muß (z.B. »MEZ« für die mitteleuropäische Zeit).

Das ebenfalls obligatorische Feld [+|-]d[d] steht für die Abweichung von der Weltstandardzeit (»Greenwich Mean Time« – GMT) in Stunden, die

S–T

durch einen Zahlenwert mit mindestens einer Ziffer (und falls notwendig, einem Vorzeichen) ausgedrückt wird. Die Verschiebung für Länder westlich des 0. Längengrades wird mit positiven Werten notiert (+5 entspricht beispielsweise der Zeitverschiebung von New York gegenüber Greenwich), für Zeitzonen östlich von Greenwich entstehen negative Werte (für die MEZ gilt beispielsweise -1). Dieser Wert wird zur Berechnung der globalen Variablen *_timezone* verwendet. *_timezone* repräsentiert die Zeitdifferenz in Sekunden zwischen der Weltstandardzeit (GMT) und der Ortszeit.

Das Feld *lll* ist im Gegensatz zu den restlichen Feldern optional: Wenn es angegeben ist, muß es aus drei Buchstaben bestehen und legt den Namen der lokalen »Sommerzeitzone« fest (z.B. »MES« für mitteleuropäische Sommerzeit). *tzset* setzt die globale Variable *_daylight* abhängig vom Vorhandensein dieses Feldes entweder auf 0 (= keine Sommerzeit) oder auf einen Wert ungleich null (Sommerzeitzone definiert).

Wenn *tzset* keine Umgebungsvariable des Namens TZ findet oder dieser Eintrag nicht den zuvor gezeigten Regeln genügt, verwendet die Funktion den String »EST5EDT« (entspricht »Eastern Standard Time«, Zeitverschiebung +5 Stunden gegenüber GMT, Sommerzeit definiert) und belegt die globalen Variablen *_daylight*, *_timezone* und *_tzname* mit den entsprechenden Werten.

Die globale Variable *_tzname[0]* zeigt auf einen drei Zeichen langen String, der den Namen der auf der Umgebungsvariable TZ ermittelten Zeitzone speichert. *_tzname[1]* zeigt auf einen drei Zeichen langen String, der den Namen der Sommerzeitzone aus TZ speichert. Wurde keine Sommerzeitzone genannt, zeigt *_tzname[1]* auf einen leeren String.

Rückgabewert Keiner.

Siehe auch *asctime*, *ctime*, *ftime*, *gmtime*, *localtime*, *stime*, *time*

ultoa — stdlib.h

Funktion Konvertiert einen Wert vom Typ **unsigned long** in einen String.

Syntax

```
char *ultoa(unsigned long value, char *string, int radix);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Funktion konvertiert den übergebenen Wert *value* in einen nullterminierten String und speichert das Ergebnis in der durch *string* bezeichneten Variablen. *ultoa* erwartet einen Wert vom Typ **unsigned long** in *value*.

Der Parameter *radix* legt die Basis fest, zu der die Konvertierung stattfinden soll – mögliche Werte liegen im Bereich von 2 bis 36. *ultoa* prüft nicht auf Überlauf; wenn *radix* den Wert 10 hat und *value* kleiner als null ist, stellt *ultoa* dem Ergebnis kein Minuszeichen voran.

➡ Der durch *string* übergebene String muß in allen Fällen lang genug sein, um das Ergebnis der Konvertierung inklusive des abschließenden Nullzeichens (\0) aufnehmen zu können. *ultoa* kann maximal 33 Zeichen zurückgeben.

Rückgabewert Die Funktion liefert *string* zurück.

Siehe auch *itoa*, *ltoa*

umask — io.h

Funktion Setzt das Bitfeld für Dateizugriffe (Lese- und Schreiboperationen).

Syntax `unsigned umask(unsigned mode);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *umask* setzt das Bitfeld, das bei Aufrufen von *open* und *creat* verwendet wird. Die in *mode* gesetzten Bits werden in den Zugriffsberechtigungen aller im folgenden mit *create* und *open* erzeugten Dateien gelöscht.

Der Parameter *mode* kann über die folgenden symbolischen Konstanten festgelegt werden, die in sys/stat.h definiert sind:

Wert von *mode*	Zugriffserlaubnis
S_IWRITE	Schreiben erlaubt
S_IREAD	Lesen erlaubt
S_IREAD\|S_IWRITE	Lesen und Schreiben erlaubt

Rückgabewert Es wird der vorherige Wert von *mask* zurückgeliefert. Ein Fehlerfall existiert nicht.

Siehe auch *creat*, *open*

ungetc — stdio.h

Funktion Stellt ein aus einem Stream gelesenes Zeichen wieder zurück.

Syntax `int ungetc(int c, FILE *stream);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung *ungetc* stellt ein Zeichen für eine erneute Leseoperation zurück in den Eingabe-Stream *stream*, der für Leseoperationen geöffnet sein muß. Ein Zeichen kann jederzeit zurückgestellt und später mit *getc* oder *fread* wieder gelesen werden. Es kann immer nur ein Zeichen zurückgestellt werden – ein zweiter Aufruf von *ungetc* ohne ein dazwischenliegendes *getc* überschreibt das zuvor zurückgestellte Zeichen. Die Anwendung von *fflush*, *fseek*, *fsetpos* oder *rewind* auf eine Datei löscht deren Puffer und damit auch ein eventuell mit *ungetc* zurückgestelltes Zeichen.

Rückgabewert *ungetc* liefert bei fehlerfreier Ausführung das zurückgestellte Zeichen, ansonsten EOF zurück.

Siehe auch *fgetc*, *getc*, *getchar*

ungetch conio.h

Funktion Stellt ein von der Tastatur gelesenes Zeichen in den Tastaturpuffer zurück.

Syntax `int ungetch(int c);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■			■

Beschreibung *ungetch* stellt das Zeichen *ch* für die nächste Leseaktion von der Tastatur zurück. Es kann nur jeweils ein Zeichen gespeichert werden – *ungetch* überschreibt ein zuvor zurückgestelltes Zeichen aber nicht, sondern liefert einen Fehler.

Rückgabewert Die Funktion liefert das Zeichen *ch* zurück; bei einem Fehler wird EOF zurückgeliefert.

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Siehe auch *getch*, *getche*

unixtodos dos.h

Funktion Konvertiert Datum und Uhrzeit aus dem UNIX- in das DOS-Format.

Syntax
```
void unixtodos(long time, struct date *d, struct time *t);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *unixtodos* erwartet einen Wert vom Typ **long**, der Datum und Uhrzeit im UNIX-Format enthält, und setzt die Strukturen des Typs *date* bzw. *time*, auf die die Parameter *d* und *t* zeigen.
time muß eine Zeitangabe nach dem 1. Januar 1980 00:00:00 darstellen.

Rückgabewert Keiner.

Siehe auch *dostounix*

unlink io.h

Funktion Löscht eine Datei.

Syntax
```
int unlink(const char *filename);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *unlink* löscht die durch *filename* angegebene Datei. Als *filename* kann jeder Laufwerksbezeichner, jeder Pfad und Dateiname verwendet werden. Die Jokerzeichen * und ? sind in *filename* nicht erlaubt.

Schreibgeschützte Dateien können mit *unlink* nicht gelöscht werden. Um eine schreibgeschützte Datei zu löschen, muß zuerst das entsprechende Attribut mit *chmod* oder *_rtl_chmod* entfernt werden.

➥ Eine geöffnete Datei muß vor dem Löschen geschlossen werden.

Rückgabewert Bei fehlerfreier Ausführung liefert *unlink* den Wert 0 zurück. Im Fehlerfall ist das Funktionsergebnis -1 und *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
ENOENT	Pfad/Datei nicht gefunden

Siehe auch *chmod*, *remove*

unlock io.h

Funktion Entfernt mit *lock* gesetzte Sperren bei Dateien im gemeinsamen Zugriff.

Syntax `int unlock(int handle, long offset, long length);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *unlock* ist eine Schnittstelle zu den Betriebssystemmechanismen für gemeinsame Dateizugriffe. *unlock* gibt einen vorher mit *lock* gesperrten Dateibereich wieder frei. Zur Vermeidung von Fehlern müssen vor dem Schließen der Datei sämtliche Bereiche entsperrt werden und ein Programm muß alle Sperren aufheben, bevor die Ausführung beendet wird.

Rückgabewert Die Funktion liefert bei fehlerfreier Ausführung den Wert 0, ansonsten den Wert -1 zurück.

Siehe auch *lock*, *locking*, *sopen*

utime — utime.h

Funktion Setzt Datum und Uhrzeit einer Datei.

Syntax `int utime(char *path, struct utimbuf *times);`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *utime* setzt Uhrzeit und Datum für die durch *path* angegebene Datei. Die Modifikationszeit ist in der durch *times* übergebenen Struktur vom Typ *utimbuf* enthalten. Diese Struktur ist in utime.h definiert und hat das folgende Format:

```
struct utimebuf {
   time_t actime;        /* Zugriffszeit */
   time_t modtime;       /* Modifikationszeit */
};
```

Das FAT Dateisystem unterstützt nur die Modifikationszeit. Im FAT-Dateisystem wird *asctime* daher bei einem Aufruf von *utime* ignoriert und nur *modtime* zum Setzen der Modifikationszeit verwendet.

Hat *times* den Wert NULL, wird die Modifikationszeit der Datei auf die aktuelle Uhrzeit gesetzt.

Rückgabewert *utime* liefert bei fehlerfreier Ausführung den Wert 0 zurück. Im Fehlerfall wird -1 zurückgeliefert und *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
EMFILE	Zu viele Dateien bereits offen
ENOENT	Pfad/Datei nicht gefunden

Siehe auch *setftime*, *stat*, *time*

va_arg, va_end, va_start — stdarg.h

Funktion Makros für Funktionen mit variabler Parameterzahl.

Syntax

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Verschiedene C-Funktionen (wie *vfprintf* und *vprintf*) erwarten einen oder mehrere obligatorische Parameter (wie ein »Ziel« und einen Format-String) und darüber hinaus eine variable Anzahl weiterer Parameter. Die Makros *va_arg*, *va_end* und *va_start* ermöglichen es, Deklarationen und Aufrufe von Funktionen mit variabler Parameterzahl in portierbarer Form zu halten. Mit ihrer Hilfe kann eine Parameterliste schrittweise durchgegangen werden, wenn die aufgerufene Funktion die Anzahl und die Typen der übergebenen Parameter nicht kennt.

In der Datei stdarg.h sind ein Typ (*va_list*) und drei Makros (*va_start*, *va_arg* und *va_end*) deklariert:

- *va_list* ist der Array-Typ, der die von *va_arg* und *va_end* benötigten Informationen speichert. Eine Funktion mit einer variablen Parameterliste deklariert eine Variable *ap* vom Typ *va_list*.
- *va_start* ist ein Makro, das *ap* auf den ersten variablen Parameter setzt, der der Funktion übergeben wird. Über *lastfix* wird der Name des letzten festen Parameters der Funktion angegeben. *va_start* muß vor *va_arg* und *va_end* aufgerufen werden.
- Das Makro *va_arg* erwartet einen (zuvor mit *va_start* initialisierten) Zeiger *ap* sowie einen Variablentyp (*type*), interpretiert den jeweils nächsten variablen Parameter der Funktion als Wert des angegebenen Typs und liefert ihn zurück. Danach wird der Zeiger *ap* um **sizeof**(*type*) erhöht.

➥ Für den Parameter *type* dürfen die Werte **char**, **unsigned char** oder **float** nicht verwendet werden.

- Der erste Aufruf von *va_arg* liefert den Wert des ersten variablen Parameters zurück und erhöht *ap* entsprechend, so daß *ap* danach auf den nächsten Parameter zeigt. Weitere Aufrufe von *va_arg* – wieder mit *ap* – liefern die jeweils nachfolgenden Parameter zurück.

U–Z

- Das Makro *va_end* hilft der aufgerufenen Funktion, einen normalen Rücksprung durchzuführen. *va_end* kann *ap* verändern, so daß vor einer erneuten Verwendung wieder *va_start* aufgerufen werden muß. *va_end* sollte aufgerufen werden, nachdem *va_arg* alle Argumente abgearbeitet hat; wird dies unterlassen, resultiert daraus unter Umständen ein undefiniertes Verhalten des Programms.

Rückgabewert *va_start* und *va_end* haben keinen Rückgabewert, *va_arg* liefert den aktuellen Parameter der Funktion zurück, wobei der Ergebnistyp durch *type* bestimmt wird.

Siehe auch *v...printf*, *v...scanf*

vfprintf — stdio.h

Funktion Formatierte Ausgabe in einen Stream.

Syntax

```
int vfprintf(FILE *stream, const char *format, va_list arglist);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Die *v...printf*-Funktionen sind eine alternative Aufrufmöglichkeit der entsprechenden *...printf*-Funktionen. Sie verhalten sich exakt wie ihre *...printf*-Gegenstücke, übernehmen jedoch anstelle einer Argumentliste einen Zeiger auf eine Liste von Argumenten als Parameter.

Weitere Details über Formatierungsanweisungen finden Sie unter *printf.*

vfprintf übernimmt einen Zeiger auf eine Reihe von Argumenten, wendet auf jedes Argument eine Formatierungsanweisung an, die in dem durch *format* gegebenen Formatstring enthalten ist, und schreibt die formatierten Daten in den durch *stream* bezeichneten Stream. Die Anzahl der Argumente und Formatierungsanweisungen muß identisch sein.

Rückgabewert Die Funktion liefert die Anzahl der ausgegebenen Bytes zurück. Im Fehlerfall liefert *vfprint* EOF zurück.

Siehe auch *printf*, *va_arg*, *va_end*, *va_start*

vfscanf — stdio.h

Funktion Formatierte Eingabe aus einem Stream.

Syntax

```
int vfscanf(FILE *stream, const char *format, va_list arglist);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung

Die *v...scanf*-Funktionen sind eine alternative Aufrufmöglichkeit der entsprechenden *...scanf*-Funktionen. Sie verhalten sich exakt wie ihre *...scanf*-Gegenstücke, übernehmen jedoch anstelle einer Argumentliste einen Zeiger auf eine Liste von Argumenten als Parameter.

Weitere Details über Formatierungsanweisungen finden Sie unter *scanf*.

vfscanf liest eine Reihe von Eingabefeldern zeichenweise aus dem durch *stream* angegebene Stream. Auf jedes Feld wird dann die entsprechende Formatierungsanweisung angewendet, die *vfscanf* über den Zeiger *format* auf den Formatstring übergeben wurde. Der Formatstring legt die Anzahl der Eingaben und ihr Format fest. Gelesene Eingaben werden an den Adressen, die in den *format* nachfolgenden Parametern angegeben werden, gespeichert. Nach der Analyse der ersten Formatierungsanweisung wird ein Eingabefeld gelesen, konvertiert und an der entsprechenden Adresse gespeichert. Die Anzahl der Eingabefelder muß mit der Anzahl der Formatierungsanweisungen und Adressen übereinstimmen.

Es ist möglich, daß *vfscanf* das Einlesen eines bestimmten Feldes vorzeitig – d.h. bevor das Feldende (markiert durch ein Whitespace-Zeichen) erreicht ist – abbricht bzw. den Lesevorgang komplett beendet. In der Beschreibung zu *scanf* werden die Gründe hierfür erläutert.

Rückgabewert

vfscanf liefert die Anzahl der fehlerfrei gelesenen und gespeicherten Felder zurück, wobei nur gelesene aber nicht gespeicherte Felder nicht gezählt werden. Wenn überhaupt keine Zuweisungen erfolgt sind, liefert *vfscanf* den Wert 0.

Beim Versuch, über das Ende einer Datei hinauszulesen, liefert *vfscanf* Wert EOF zurück.

Siehe auch

fscanf, *scanf*, *va_arg*, *va_end*, *va_start*

vprintf — stdarg.h

Funktion

Formatierte Ausgabe nach *stdout*.

Syntax

```
int vprintf(const char *format, va_list arglist);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■		■	■		■

Beschreibung

Die *v...printf*-Funktionen sind eine alternative Aufrufmöglichkeit der entsprechenden *...printf*-Funktionen. Sie verhalten sich exakt wie ihre *...printf*-Gegenstücke, übernehmen jedoch anstelle einer Argumentliste einen Zeiger auf eine Liste von Argumenten als Parameter.

Weitere Details über Formatierungsanweisungen finden Sie unter *printf*.

vprintf akzeptiert einen Zeiger auf ein Reihe von Argumenten, wendet auf jedes Argument eine Formatierungsanweisung an, die in dem durch *format* angegebenen Formatstring enthalten ist, und schreibt die formatierten Daten nach *stdout*. Die Anzahl der Argumente und muß mit der der Formatierungsanweisungen übereinstimmen.

➡ Wenn Sie das Flag *SS!=DS* bei 16-Bit Anwendungen verwenden, dann geht *vprintf* davon aus, daß sich die übergebene Adresse im SS-Segment befindet.

➡ In Win32 oder Win32s GUI-Anwendungen muß *stdout* umgelenkt werden.

Rückgabewert Die Funktion liefert die Anzahl der ausgegebenen Zeichen zurück. Bei einem Fehler liefert *vprintf* den Wert EOF zurück.

Siehe auch *freopen*, *printf*, *va_arg*, *va_end*, *va_start*

vscanf — stdarg.h

Funktion Formatierte Eingabe von *stdin*.

Syntax

```
int vscanf(const char *format, va_list arglist);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung Die *v...scanf*-Funktionen sind eine alternative Aufrufmöglichkeit der entsprechenden *...scanf*-Funktionen. Sie verhalten sich exakt wie ihre *...scanf*-Gegenstücke, übernehmen jedoch anstelle einer Argumentliste einen Zeiger auf eine Liste von Argumenten als Parameter.

Weitere Details über Formatierungsanweisungen finden Sie unter *scanf*.

vscanf liest eine Reihe von Eingabefeldern zeichenweise aus *stdin*. Auf jedes Feld wird dann die entsprechende Formatierungsanweisung angewendet, die *vscanf* über den Zeiger *format* auf den Formatstring übergeben wurde. Der Formatstring legt die Anzahl der Eingaben und ihr Format fest. Gelesene Eingaben werden an den Adressen gespeichert, die in den *format* nachfolgenden Parametern angegeben werden. Nach der Analyse der ersten Formatierungsanweisung wird ein Eingabefeld gelesen, konvertiert und an der entsprechenden Adresse gespeichert. Die Anzahl der Eingabefelder muß mit der Anzahl der Formatierungsangweisungen und Adressen übereinstimmen.

Es ist möglich, daß *vscanf* das Einlesen eines bestimmten Feldes vorzeitig – d.h. bevor das Feldende (markiert durch ein Whitespace-Zeichen) erreicht ist – abbricht bzw. den Lesevorgang komplett beendet. In der Beschreibung zu *scanf* werden die Gründe hierfür erläutert.

Rückgabewert *vscanf* liefert die Anzahl der fehlerfrei gelesenen, konvertierten und gespeicherten Felder zurück, wobei Felder, die nur gelesen aber nicht gespeichert wurden, nicht gezählt werden. Wenn überhaupt keine Zuweisungen erfolgt sind, ist der Rückgabewert 0.

Beim Versuch, über das Ende der Datei hinauszulesen, liefert *vscanf* den Wert EOF zurück.

Siehe auch *freopen*, *fscanf*, *scanf*, *va_arg*, *va_end*, *va_start*

vsprintf — stdarg.h

Funktion Formatierte Ausgabe in einen String.

Syntax

```
int vsprintf(char *buffer, const char *format, va_list arglist);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Die *v...printf*-Funktionen sind eine alternative Aufrufmöglichkeit der entsprechenden *...printf*-Funktionen. Sie verhalten sich exakt wie ihre *...printf*-Gegenstücke, übernehmen jedoch anstelle einer Argumentliste einen Zeiger auf eine Liste von Argumenten als Parameter.

Weitere Details über Formatierungsanweisungen finden Sie unter *printf*.

vsprintf akzeptiert einen Zeiger auf ein Reihe von Argumenten, wendet auf jedes Argument eine Formatierungsanweisung an, die in dem durch *format* angegebenen Formatstring enthalten ist, und schreibt die formatierten Daten in den durch *buffer* bezeichneten String. Die Anzahl der Argumente muß mit der der Formatierungsanweisungen übereinstimmen.

Rückgabewert Die Funktion liefert die Anzahl der ausgegebenen Zeichen zurück. Bei einem Fehler liefert *vsprintf* EOF zurück.

Siehe auch *printf*, *va_arg*, *va_end*, *va_start*

vsscanf — stdarg.h

Funktion Formatierte Eingabe aus einem String.

Syntax

```
int vsscanf(const char *buffer, const char *format, va_list arglist);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

U–Z

Beschreibung Die *v...scanf*-Funktionen sind eine alternative Aufrufmöglichkeit der entsprechenden *...scanf*-Funktionen. Sie verhalten sich exakt wie ihre *...scanf*-

Gegenstücke, übernehmen jedoch anstelle einer Argumentliste einen Zeiger auf eine Liste von Argumenten als Parameter.

Weitere Details über Formatierungsanweisungen finden Sie unter *scanf.*

vsscanf liest eine Reihe von Eingabefeldern zeichenweise aus dem durch *buffer* bezeichneten String. Auf jedes Feld wird dann die entsprechende Formatierungsanweisung angewendet, der *vsscanf* über den Zeiger *format* auf den Formatstring übergeben wurde. Der Formatstring legt die Anzahl der Eingaben und ihr Format fest. Gelesene Eingaben werden an den Adressen gespeichert, die in den *format* nachfolgenden Parametern angegeben werden. Nach der Analyse der ersten Formatierungsanweisung wird ein Eingabefeld gelesen, konvertiert und an der entsprechenden Adresse gespeichert. Die Anzahl der Eingabefelder muß mit der Anzahl der Formatierungsanweisungen und Adressen übereinstimmen.

Es ist möglich, daß *vsscanf* das Einlesen eines bestimmten Feldes vorzeitig – d.h. bevor das Feldende (markiert durch ein Whitespace-Zeichen) erreicht ist – abbricht bzw. den Lesevorgang komplett beendet. In der Beschreibung zu *scanf* werden die Gründe hierfür erläutert.

Rückgabewert

vsscanf liefert die Anzahl der fehlerfrei gelesenen, konvertierten und gespeicherten Felder zurück, wobei Felder, die zwar gelesen aber nicht gespeichert wurden, nicht gezählt werden. Wenn überhaupt keine Zuweisungen erfolgt sind, ist der Rückgabewert 0.

Beim Versuch, über das Ende des Strings hinauszulesen, wird der Wert EOF zurückgeliefert.

Siehe auch

fscanf, *scanf*, *sscanf*, *va_arg*, *va_end*, *va_start*, *vfscanf*

wait — process.h

Funktion

Wartet auf das Ende von Kindprozessen.

Syntax

```
int wait(int *statloc);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
			■			■

Beschreibung

Die Funktion *wait* wartet auf das Ende eines oder mehrerer Kindprozesse. Die Kindprozesse müssen vom aufrufenden Programm selbst erzeugt worden sein. *wait* kann nicht auf Enkelprozesse (Prozesse die von Kindprozessen erzeugt wurden) warten. Enthält der Parameter *statloc* einen Wert ungleich NULL, dann wird an dieser Adresse der Beendigungs-Status des Kindprozesses gespeichert.

Bei normaler Beendigung des Kindprozesses (durch Aufruf von *exit* oder Rücksprung von *main*) ist das Statuswort folgendermaßen definiert:

Bit 0-7 Null

Bit 8-15 Das niederwertige Byte des Rückgabewerts des Kindprozesses. Das ist der durch *exit* oder den Rücksprung aus *main* erhaltene Wert. Wird der Kindprozeß durch einfachen Rücksprung aus *main* (ohne Wert) verlassen, so ist der Inhalt dieser Bits unbestimmt.

Wird der Kindprozeß nicht normal beendet, dann ist das Statuswort wie folgt definiert:

Bits 0-7 Information über die Beendigung des Kindprozesses
1 Abbruch durch kritischen Fehler
2 Ausführungfehler, Schutzverletzung
3 Externes Signal

Bits 8-15 Null

Rückgabewert

Bei einer normalen Beendigung des Kindprozesses liefert *wait* die Prozeß-ID des Kindes zurück.

Bei einer anormalen Beendigung des Kindprozesses liefert die Funktion -1 an den Elternprozeß zurück und setzt *errno* auf EINTR.

Wenn *wait* zurückkehrt, ohne daß ein Kindprozeß beendet wurde, so liefert die Funktion -1 zurück und setzt *errno* auf:

ECHILD Kein Kindprozeß vorhanden

Siehe auch

cwait, *spawn*

wcstombs — stdlib.h

Funktion

Konvertiert ein *wchar_t*-Array in einen String aus Multibyte-Zeichen.

Syntax

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung

wcstombs konvertiert die über *pcws* angegebenen Elemente des Typs *wchar_t* in den Multibyte-String *s*. Die Konvertierung wird beendet, wenn *wcstombs* auf ein Nullzeichen oder ein ungültiges Multibyte-Zeichen trifft.

Der Parameter *n* legt fest, wieviele Bytes maximal konvertiert werden. Wenn die betreffende Anzahl Bytes konvertiert und kein Nullzeichen angetroffen wurde, ist das Array *s* nicht nullterminiert.

Das Verhalten von *wcstombs* ist von der Kategorie LC_CTYPE der aktuellen *locale*-Einstellung abhängig.

U–Z

Rückgabewert *wcstombs* liefert *(size_t)-1* zurück, wenn es auf ein ungültiges Multibyte-Zeichen trifft. Andernfalls wird die Anzahl der geänderten Bytes zurückgeliefert (das abschließende Zeichen ist – falls vorhanden – darin nicht enthalten).

wctomb stdlib.h

Funktion Konvertiert einen Wert des Typs *wchar_t* in ein Multibyte-Zeichen.

Syntax

```
int wctomb(char *s, wchar_t wc);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Beschreibung Ist *s* ungleich NULL, stellt *wctomb* die Anzahl der Bytes fest, die zur Darstellung des durch *wc* angegebenen Multibyte-Zeichens erforderlich sind (einschließlich vom Status der Umschalttasten abhängiger Codierungen). Das Multibyte-Zeichen wird in *s* gespeichert. Maximal werden MB_CUR_MAX gespeichert. Hat *wc* den Wert Null, behält *wctomb* den ursprünglichen Zustand bei.

Das Verhalten von *wctomb* ist von der Kategorie LC_CTYPE der aktuellen *locale*-Einstellung abhängig.

Rückgabewert Ist *s* ein NULL-Zeiger, so liefert *wctomb* einen Wert ungleich null, wenn die Multibyte-Zeichen statusabhängige Codierungen enthalten. Sind keine statusabhängigen Codierungen enthalten, so wird null zurückgeliefert.

Ist *s* kein NULL-Zeiger, so liefert *wctomb* den Wert -1 zurück, wenn *wc* kein gültiges Multibyte-Zeichen enthält. Ansonsten liefert *wctomb* die Anzahl der Bytes des durch *wc* angegebenen Multibyte-Zeichens zurück. Der Rückgabewert ist nie größer als der Wert von MB_CUR_MAX.

wherex conio.h

Funktion Liefert die Spaltenposition des Cursors innerhalb eines Fensters zurück.

Syntax

```
int wherex(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *wherex* liefert die aktuelle Spalte (die x-Koordinate) der Cursor-Position zurück. Die Funktion arbeitet relativ zum aktuellen Textfenster.

➡ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert *wherex* liefert einen Integerwert im Bereich von 1 bis zur Anzahl der Spalten im aktuellen Videomodus zurück.

Siehe auch *gettextinfo*, *gotoxy*, *wherey*

wherey conio.h

Funktion Liefert die aktuelle Zeile der Cursor-Position innerhalb eines Fensters zurück.

Syntax

```
int wherey(void);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *wherey* liefert die aktuelle Zeile (die y-Koordinate) der Cursor-Position zurück. Die Funktion arbeitet relativ zum aktuellen Textfenster.

➡ Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert *wherey* liefert einen Integerwert im Bereich von 1 bis zur Anzahl der Zeilen im aktuellen Videomodus zurück.

Siehe auch *gettextinfo*, *gotoxy*, *wherex*

window conio.h

Funktion Definiert einen Bereich des Bildschirms als Textfenster.

Syntax

```
void window(int left, int top, int right, int bottom);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung *window* definiert einen rechteckigen Bereich des Bildschirms als Textfenster, dessen obere linke Ecke mit (*left*, *top*) und dessen untere rechte Ecke durch (*right*, *bottom*) festgelegt ist. Die obere linke Ecke des Bildschirms hat die Koordinaten (1,1), die Minimalgröße des Fensters ist ein einzelner Buchstabe (d.h. ein Spalte und eine Zeile).

Wenn einer der vier Parameter von *window* nicht innerhalb der erlaubten Grenzen liegt, wird der Aufruf ignoriert und das aktuelle Fenster bleibt unverändert.

Die Minimalgröße des Textfensters beträgt 1 Zeile mal 1 Spalte. Die Standardgröße entspricht der vollen Bildschirmgröße mit den Koordinaten

```
1,1,Sp,Ze
```

wobei `Sp` die Anzahl der Spalten und `Ze` die Anzahl der Zeilen im aktuellen Videomodus bezeichnet.

Diese Funktion sollte in Win32s oder Win32 GUI-Anwendungen nicht verwendet werden.

Rückgabewert Keiner.

Siehe auch *clreol*, *clrscr*, *delline*, *gettextinfo*, *gotoxy*, *insline*, *puttext*, *textmode*

_write — io.h

Veraltete Funktion. Siehe *_rtl_write*.

write — io.h

Funktion Schreibt Daten in eine Datei.

Syntax

```
int write(int handle, void *buf, unsigned len);
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■			■

Beschreibung *write* liest *len* Bytes ab der durch *buf* angegebenen Adresse aus dem Speicher und schreibt sie in die durch *handle* bezeichnete Datei. *handle* ist ein Datei-Handle, das man durch einen Aufruf von *creat*, *open*, *dup* oder *dup2* erhält. Mit Ausnahme von Textdateien entspricht die Anzahl der in die Datei geschriebenen Bytes dem durch den Parameter *len* angegebenen Wert. Mit *write* können maximal UINT_MAX-1 Bytes geschrieben werden, da UINT_MAX dem Wert -1 entspricht, dem Fehlerindikator von *write*. UINT_MAX ist in limits.h definiert. Wenn die Datei im Textmodus geöffnet wurde und *write* auf das Zeilenvorschubzeichen LF ('\n') trifft, so wird die Zeichenfolge CR/LF ausgegeben.

Ist die Anzahl der tatsächlich geschriebenen Bytes kleiner als *len*, so ist dies ein Fehler und bedeutet wahrscheinlich, daß die Diskette/Festplatte voll ist. Bei Diskettendateien schreibt *write* ab der aktuellen Position des Dateizeigers, bei Gerätedateien wird direkt zum jeweiligen Gerät geschrieben. Wenn die Datei mit O_APPEND (anhängen) geöffnet wurde, setzt *write* den Dateizeiger automatisch auf das Dateiende.

Rückgabewert

Die Funktion liefert die Anzahl der tatsächlich geschriebenen Bytes zurück (wobei *write* die aus Zeilenvorschüben erzeugten CR/LF-Kombinationen als ein Byte zählt). Im Fehlerfall wird der Wert -1 zurückgeliefert und *errno* erhält einen der folgenden Werte:

EACCES	Zugriff nicht gestattet
EBADF	Ungültiges Handle

Siehe auch

creat, *lseek*, *open*, *read*, *_rtl_write*

U–Z

K A P I T E L **4**

Globale Variablen

Borland C++ Compiler stellen Ihnen für viele allgemeine Standardgrößen vordefinierte globale Variablen wie Kalenderdatum und Uhrzeit, Kommandozeilenparameter usw. zur Verfügung. In diesem Kapitel werden die globalen Variablen beschrieben und definiert.

_8087 dos.h

Funktion Koprozessor-Flag.

Syntax `extern int _8087;`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung Diese Variable wird durch die automatische Erkennung des Startcodes auf einen Wert ungleich 0 (1, 2 oder 3) gesetzt, wenn ein installierter mathematischer Koprozessor (ein 8087, 80287 oder ein 80387) erkannt wurde. Andernfalls wird *_8087* auf den Wert 0 gesetzt.

In einem 16-Bit Windows-Programm ist der Wert gleich 1, wenn ein Koprozessor entdeckt wird.

Die automatische Erkennungslogik kann außer Kraft gesetzt werden, indem der Umgebungsvariablen 87 der Wert `YES` oder `NO` zugewiesen wird. Die Befehle hierzu sind: `SET 87= YES` oder `SET 87=NO`; vor und nach dem Gleichheitszeichen darf kein Leerzeichen stehen. Ist die Umgebungsvariable korrekt gesetzt, so wird die globale Variable *_8087* entsprechend gesetzt.

Im Kapitel 8 des *Programmierhandbuchs* finden Sie weitere Informationen über die Umgebungsvariable 87.

_argc dos.h

Funktion Zahl der Kommandozeilenparameter.

Syntax

```
extern int _argc;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_argc* enthält die als *argc* an *main* übergebene Anzahl der Kommandozeilenparameter.

_argv dos.h

Funktion Ein Array mit Zeigern auf die einzelnen Kommandozeilenparameter.

Syntax

```
extern char **_argv;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_argv* zeigt auf ein String-Array mit den einzelnen Kommandozeilenparametern, die *main* beim Programmstart als Elemente von *argv*[] übergeben wurden.

_ctype ctype.h

Funktion Ein Array mit Zeichenattribut-Informationen.

Syntax

```
extern char _ctype[];
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_ctype* ist ein Array mit Zeichenattribut-Informationen. Der Index ist der ASCII- Wert + 1. Jeder Eintrag ist ein Satz von Bits, die das Zeichen beschreiben.

Dieses Array wird nur von Routinen verwendet, die von der locale-Einstellung »C« betroffen sind (wie etwa *isdigit*, *isprint* usw.)

_daylight time.h

Funktion Legt fest, ob Anpassungen an die Sommerzeit durchgeführt werden.

Syntax `extern int _daylight;`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Variable wird zusammen mit den Funktionen für Uhrzeit und Datum verwendet. *_daylight* muß von *tzset*, *ftime* oder *localtime* auf den Wert 1 gesetzt werden, wenn die Sommerzeit berücksichtigt werden soll; 0 steht für die Standardzeit.

_directvideo conio.h

Funktion Flag für die Steuerung von Bildschirmausgaben.

Syntax `extern int _directvideo;`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			

Beschreibung *_directvideo* steuert, ob Bildschirmausgaben Ihres Programms (z.B. von *cputs*) direkt in den Bildschirmspeicher schreiben (*_directvideo* = 1) oder BIOS-Funktionen verwenden (*_directvideo* = 0).

Die Voreinstellung *_directvideo* = 1 (die Ausgabe schreibt in den Bildschirmspeicher) sollte nur verwendet werden, wenn die Video-Hardware Ihres Systems 100% kompatibel zu einer IBM-Grafikkarte ist. Mit *_directvideo = 0 benötigen Sie nur ein System, das IBM-BIOS-kompatibel ist.*

_directvideo sollte nur in Textanwendungen, nicht aber in 16-Bit Windows, Win32s oder Win32 GUI-Anwendungen verwendet werden.

_environ dos.h

Funktion Zugriff auf die Umgebungsvariablen des Betriebssystems.

Syntax `extern char **_environ;`

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

Die Variable *_environ* ist ein Array von Zeigern auf Strings, über die Umgebungsvariablen des Betriebssystems gelesen und geändert werden können. Jeder String hat die Form

```
U_Var = Wert
```

wobei *U_Var* der Name einer Umgebungsvariable (wie z.B. PATH) und *Wert* der Stringwert (wie z.B. C:\BIN;C:\DOS) ist, der *U_Var* zugewiesen wird. Der String *Wert* kann auch leer sein.

Beim Start eines Programms werden die Umgebungsvariablen des Betriebssystems direkt an das Programm übergeben. Der Inhalt von *env*, dem dritten Parameter von *main*, stimmt mit dem Anfangswert von *_environ* überein.

Auf das Array der Umgebungsvariablen kann auch über *getenv* zugegriffen werden. Für Änderungen, Löschungen oder Hinzufügungen von Umgebungsvariablen sollten Sie jedoch ausschließlich die Funktion *putenv* verwenden, da sich die Größe und die Adresse des Array ändern kann, *_environ* jedoch automatisch angepaßt wird und stets auf das Array zeigt.

Siehe auch

getenv, putenv

errno, _doserrno, _sys_errlist, _sys_nerr — dos.h, errno.h

Funktion

Ermöglicht *perror* die Ausgabe von Fehlermeldungen.

Syntax

```
extern int errno;
extern int _doserrno;
extern char **_sys_errlist;
extern int _sys_nerr;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

errno, *_sys_errlist* und *_sys_nerr* werden von *perror* zur Ausgabe von Fehlermeldungen benutzt, wenn bestimmte Bibliotheksfunktionen ihre Aufgabe nicht ausführen können. Die Variable *_doserrno* bildet viele Fehlercodes des Betriebssystems auf *errno* ab; *_doserrno* wird von *perror* aber nicht direkt benutzt. In den Header-Dateien winbase.h und winerror.h finden Sie eine vollständige Liste aller Betriebssystemfehler.

- *errno*: Bei einem Fehler nach einem Aufruf einer mathematischen Funktion oder einer Systemfunktion wird *errno* gesetzt, um den Fehlertyp anzuzeigen. Manchmal sind *errno* und *_doserrno* äquivalent, in anderen Fällen enthält *errno*, anders als *_doserrno*, nicht den aktuellen Fehlercode des Betriebssystems. Bestimmte Fehler setzen nur *errno*, nicht aber *_doserrno*.

- *_doserrno*: Wenn ein Systemaufruf des Betriebsystems mit einem Fehler endet, wird der entsprechende Fehlercode in *_doserrno* gespeichert. Die zusätzlich und gleichzeitig mitgeführte Fehlervariable *errno* stammt von UNIX.
- *_sys_errlist*: Das Array mit den Strings, die die Fehlermeldungen enthalten, steht Ihnen über die Variable *_sys_errlist* zur Verfügung und erleichtert Ihnen so die Formatierung von Meldungen. Benutzen Sie *errno* als Index in dieses Array, um den zur aktuellen Fehlernummer gehörenden String aufzufinden. Diese Strings enthalten keine Zeilenvorschubzeichen (\n).
- *_sys_nerr*: Diese Variable enthält die Anzahl der Fehlermeldungsstrings in *_sys_errlist*.

Folgende Symbole und Fehlermeldungen sind in *_sys_errlist* enthalten:

Symbol	16-Bit Beschreibung	32-Bit Beschreibung
E2BIG	Argumentenliste zu lang	Argumentenliste zu lang
EACCES	Zugriff nicht gestattet	Zugriff nicht gestattet
EBADF	Ungültiges Datei-Handle	Ungültiges Datei-Handle
ECHILD		Kein Kindprozeß
ECONTR	Speicherblöcke zerstört	Speicherblöcke zerstört
ECURDIR	Versuch, das akt. Verzeichnis zu löschen	Versuch, das akt. Verzeichnis zu löschen
EDEADLOCK		Fehler beim Sperren
EDOM	Bereichsfehler	Ungültiges Argument für mathematische Funktion
EEXIST	Datei existiert bereits	Datei existiert bereits
EFAULT	Unbekannter Fehler	Unbekannter Fehler
EINTR		Funktionsaufruf unterbrochen
EINVACC	Ungültiger Zugriffscode	Ungültiger Zugriffscode
EINVAL	Ungültiges Argument	Ungültiges Argument
EINVDAT	Ungültige Daten	Ungültige Daten
EINVDRV	Ungültiges Laufwerk angegeben	Ungültiges Laufwerk angegeben
EINVENV	Ungültiges Environment	Ungültiges Environment
EINVFMT	Ungültiges Format	Ungültiges Format
EINVFNC	Ungültige Funktionsnummer	Ungültige Funktionsnummer
EINVMEM	Ungültige Speicherblockadresse	Ungültige Speicherblockadresse
EIO		Ein-/Ausgabefehler
EMFILE	Zu viele Dateien bereits offen	Zu viele Dateien bereits offen
ENAMETOOLONG		Dateiname zu lang
ENFILE		Zu viele offene Dateien
ENMFILE	Keine weiteren Dateieinträge	Keine weiteren Dateieinträge
ENODEV	Angegebenes Gerät existiert nicht	Angegebenes Gerät existiert nicht
ENOENT	Datei/Verzeichnis nicht gefunden	Datei/Verzeichnis nicht gefunden
ENOEXEC	Exec-Formatfehler	Exec-Formatfehler
ENOFILE	Datei/Verzeichnis nicht gefunden	Datei nicht gefunden
ENOMEM	Nicht genug Platz im Hauptspeicher	Nicht genug Platz im Hauptspeicher
ENOPATH	Pfad nicht gefunden	Pfad nicht gefunden
ENOSPC		Kein Platz mehr auf Gerät
ENOTSAM	Nicht dasselbe Gerät	Nicht dasselbe Gerät
ENXIO		Gerät oder Adresse nicht gefunden
EPERM		Operation nicht erlaubt
EPIPE		Pipe zerstört
ERANGE	Ergebnis außerhalb des Rechenbereichs	Ergebnis zu groß

EROFS		Nur Lesezugriffe im Dateisystem
ESPIPE		Ungültige Suchoperation
EXDEV	»Rename«mit verschiedenen Laufwerken	»Rename«mit verschiedenen Laufwerken»
EZERO	Kein Fehler	Kein Fehler

Für direkt von DOS zurückgelieferte Fehler (d.h. mögliche Werte von *_doserrno*) sind die folgenden Konstanten in errno.h definiert. Diese Werte können teilweise über *errno* auf entsprechende Fehlerstrings aus *_sys_errlist* abgebildet werden.

Symbol	DOS-Fehlercode	Beschreibung
E2BIG	*Bad environ*	Ungültiges Environment
EACCES	*Access denied*	Zugriff verweigert
EACCES	*Bad access*	Falscher Zugriff
EACCES	*Is current dir*	Aktuelles Verzeichnis
EBADF	*Bad handle*	Ungültiges Handle
EFAULT	*Reserved*	Reserviert
EINVAL	*Bad data*	Ungültige Daten
EINVAL	*Bad function*	Ungültige Funktion
EMFILE	*Too many open*	Zu viele Dateien geöffnet
ENOENT	*No such file or _directory*	Datei/Verzeichnis nicht gefunden
ENOEXEC	*Bad format*	Ungültiges Format
ENOMEM	*Out of memory*	Nicht ausreichend Speicher
ENOMEM	*Bad block*	Ungültiger Block
EXDEV	*Bad drive*	Ungültiges Laufwerk
EXDEV	*Not same device*	Ziel- und Quellgerät sind nicht identisch

Eine detaillierte Beschreibung der Fehlerwerte von DOS finden Sie im DOS-Referenzhandbuch.

_floatconvert — stdio.h

Funktion Linkt Konvertierungsroutinen für Gleitkommaformate hinzu.

Syntax

```
extern int _floatconvert;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Ausgabe von Gleitkommawerten erfordert das Hinzulinken der von *printf*, *scanf* etc. verwendeten Konvertierungsroutinen. Zur Verringerung der Programmgröße werden diese Konvertierungsroutinen nicht automatisch gelinkt. Dies geschieht nur dann, wenn Ihr Programm eine mathematische Routine benutzt oder die Adresse einer Gleitkommazahlen ausgewertet wird. Ist beides nicht der Fall, dann können die fehlenden Konvertierungsroutinen zu einem Laufzeitfehler führen.

_fmode fcntl.h

Funktion Bestimmt den Modus (Text oder binär) beim Öffnen von Dateien.

Syntax

```
extern int _fmode;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_fmode* legt fest, ob Dateien im Text- oder im Binärmodus geöffnet und übersetzt werden. Die Voreinstellung für *_fmode* ist O_TEXT, wodurch Dateien im Textmodus gelesen werden. Um Dateien im Binärmodus zu öffnen und zu lesen, muß *_fmode* auf O_BINARY gesetzt werden. Die Konstanten O_TEXT und O_BINARY sind in fcntl.h definiert.

Im Textmodus werden bei der Eingabe CR/LF-Kombinationen (Wagenrücklauf/Zeilenvorschub) in ein einzelnes LF-Zeichen übersetzt. Bei der Ausgabe werden umgekehrt einzelne LF-Zeichen in CR/LF-Kombinationen übersetzt.

Im Binärmodus findet keinerlei Übersetzung statt.

Der durch *_fmode* voreingestellte Modus kann durch Angabe eines *t* (für Textmodus) oder *b* (für Binärmodus) als Parameter *type* bei den Bibliotheksfunktionen *fopen*, *fdopen* und *freopen* geändert werden. Auch beim Aufruf der Funktion *open* kann durch Angabe einer der beiden Konstanten O_BINARY oder O_TEXT im Parameter *access* Binärmodus bzw. Textmodus für die zu öffnende Datei explizit gewählt werden.

_new_handler

Funktion Fängt Fehler bei der Reservierung von neuem Speicherplatz ab.

Syntax

```
typedef void(*pvf)();
pvf _new_handler;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Alternativ können Sie auch die Funktion *set_new_handler* wie folgt verwenden:

```
pvf set_new_handler(pvf p);
```

_new_handler beinhaltet einen Zeiger auf eine Funktion, der keine Parameter übergeben werden und die einen Rückgabewert vom Typ **void** hat. Wenn der Operator *new()* den benötigten Speicherplatz nicht reservieren

kann, wird die Funktion aufgerufen, auf die *_new_handler* zeigt. Nach der Rückkehr von dieser Funktion wird erneut versucht, den Speicherplatz zu reservieren. Per Voreinstellung weist *_new_handler* auf eine Funktion, die das Programm beendet. Das Programm kann diesen Handler durch eine Funktion ersetzen, die versucht, Speicherplatz freizugeben. Dies kann über eine direkte Zuweisung an *_new_handler* oder durch den Aufruf der Funktion *_set_new_handler* geschehen. *set_new_handler* liefert einen Zeiger auf den früheren Handler zurück.

_new_handler wird primär aus Kompatibilitätsgründen zur C++ Version 1.2 von AT&T bereitgestellt. In den meisten Fällen erreicht man diese Funktionalität besser durch Überladen des Operators *new()*.

_osmajor, _osminor, _osversion — dos.h

Funktion Haupt- und Neben-Versionsnummer des Betriebssystems.

Syntax

```
extern unsigned char _osmajor;
extern unsigned char _osminor;
extern unsigned _osversion;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Haupt- und Neben-Versionsnummern des Betriebssystems stehen über *_osmajor* und *_osminor* zur Verfügung. *_osmajor* beinhaltet die Haupt-Versionsnummer, *_osminor* die Neben-Versionsnummer. Bei Verwendung der DOS-Version 3.2 ist beispielsweise der Wert von *_osmajor* auf 3, der Wert von *_osminor* auf 20 gesetzt.

_osversion ist funktional mit *_version* identisch. Siehe Beschreibung auf Seite 334.

Diese Variablen sind hilfreich, wenn ein Programm unter den DOS-Versionen 2.x und 3.x laufen soll. Einige Bibliotheksfunktionen verhalten sich in Abhängigkeit von der DOS-Versionsnummer unterschiedlich, andere funktionieren nur unter DOS 3.x. Vergleichen Sie hierzu beispielsweise die Beschreibungen von *open*, *creatnew* und *ioctl* in diesem Buch.

_psp — dos.h

Funktion Enthält die Segmentadresse des Programmsegmentpräfix (PSP) des aktuellen Programms.

Syntax

```
extern unsigned int _psp;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

Die Variable *_psp* enthält die (normalisierte) Segmentadresse des Programmsegmentpräfix beim Start eines Programms – eines von DOS reservierten Bereichs, der interne Informationen wie die höchste verfügbare Speicheradresse, Kommandozeilenparameter und anderes enthält.

Ausführliche Informationen zum PSP finden Sie im DOS-Referenzhandbuch.

_threadid stddef.h

Funktion

Zeiger auf Thread-ID.

Syntax

```
extern long _threadid;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■			■			■

Beschreibung

_threadid enthält die ID des aktuell ausgeführten Thread. *_threadid* ist als Makro implementiert und sollte nur durch Einbinden der Header-Datei stddef.h deklariert werden.

__throwExeceptionName, __throwFileName, __throwLineNumber except.h

Funktion

Erzeugt Informationen über ausgelöste Fehler.

Syntax

```
extern char * __throwExeptionName;
extern char * __throwFileName;
extern char * __throwLineNumber;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung

Verwenden Sie diese globalen Variablen, um den Namen und die Position einer ausgelösten Exception zu ermitteln. Jede Variable enthält einen druckbaren String.

Damit der Dateiname und die Zeilennummer über *__throwFileName* und *__throwLineNumber* ermittelt werden kann, muß das Modul mit der Compileroption -xp compiliert werden.

_timezone time.h

Funktion Enthält den Unterschied zwischen der lokalen Zeit und der GMT in Sekunden.

Syntax

```
extern long _timezone;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Variable wird von den Funktionen für Uhrzeit und Datum verwendet.

_timezone wird von der Funktion *tzset* berechnet. *_timezone* ist ein **long**-Wert, der den Unterschied zwischen der Greenwich Mean Time und der lokalen Zeit in Sekunden wiedergibt.

Siehe auch *_daylight*

tzname time.h

Funktion Ein Array von Zeigern auf die Namen der Zeitzonen.

Syntax

```
extern char *tzname[2];
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die globale Variable *tzname* ist ein Array von Zeigern auf Strings, die Abkürzungen von Zeitzonennamen enthalten. *tzname[0]* zeigt auf einen String mit drei Zeichen, der den Zeitzonennamen aus der Umgebungsvariable TZ enthält. *tzname[1]* zeigt auf einen String mit drei Zeichen, der den Zeitzonennamen der Sommerzeit aus der Umgebungsvariable TZ enthält. Wenn die Sommerzeit-Konvertierung nicht benutzt wird, zeigt *tzname[1]* auf einen leeren String.

_version dos.h

Funktion Enthält die Versionsnummer des Betriebssystems.

Syntax

```
extern unsigned _version;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung *_version* enthält die Nummer der Betriebssystemversion, die Haupt-Versionsnummer im höherwertigen Byte und die Neben-Versionsnummer im niederwertigen Byte. Bei 32-Bit Anwendungen bezieht sich diese Anordnung auf das niederwertige Wort. Bei der DOS-Version *x.y* ist *x* die Haupt- und *y* die Neben-Versionsnummer.

_wscroll — conio.h

Funktion Ein- oder Abschalten des Bildlaufs bei Bildschirm-Ein-/Ausgabefunktionen.

Syntax

```
extern int _wscroll;
```

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■		■	■			■

Beschreibung Die Variable *_wscroll* ist ein Flag für Bildschirm-Ein-/Ausgabefunktionen. Die Voreinstellung ist 1. Setzen Sie *_wscroll* auf 0, so ist der Bildlauf ausgeschaltet. Damit verhindern Sie unerwünschten Bildlauf, wenn Sie Zeichen in der unteren rechten Ecke eines Fensters ausgeben wollen.

_wscroll sollte nur in Textanwendungen, nicht aber in 16-Bit Windows, Win32s oder Win32 GUI-Anwendungen verwendet werden.

K A P I T E L 5

Die C++ Klassen für Ein-/Ausgabe-Streams

In der Online-Hilfe finden Sie Beispielprogramme für viele Ein-/Ausgabe-Stream-Klassen

Die Stream-Klassenbibliothek von C++ besteht aus mehreren Klassen, die auf zwei Hierarchiebäume verteilt sind. Im Kapitel 6 des *Programmierhandbuch* finden Sie eine Abbildung der Klassenhierarchien. In dieser Referenz finden Sie in alphabetischer Reihenfolge die wichtigsten Informationen über diese Klassen. In der folgenden Tabelle sind die einzelnen Klassen ihren Header-Dateien zugeordnet.

Header-Datei	Klassen
constrea.h	*conbuf*, *constream* (Diese Klassen sind nur für Textbildschirmanwendungen verfügbar)
iostream.h	*ios*, *iostream*, *iostream_withassign*, *istream*, *istream_withassign*, *ostream*, *ostream_withassign*, *streambuf*
fstream.h	*filebuf*, *fstream*, *fstreambase*, *ifstream*, *ofstream*
strstrea.h	*istrstream*, *ostrstream*, *strstream*, *strstreambase*, *strstreambuf*

Klasse conbuf — constream.h

conbuf ist nur für Textbildschirmanwendungen verfügbar.

Spezialisierung von *streambuf* für die Verwaltung von Textbildschirmausgaben.

Public-Konstruktoren

```
conbuf();
```

Erzeugt ein neues, nicht verbundenes *conbuf*-Objekt.

Public-Elementfunktionen

clreol `void clreol();`

Löscht in einem Textfenster den Text bis zum Ende der Zeile.

clrscr `void clrscr();`

Löscht den definierten Bildschirm.

delline `void delline();`

Löscht eine Zeile im Textfenster.

gotoxy `void gotoxy(int x, int y);`

Setzt den Cursor im Textfenster an die angegebene Position.

highvideo `void highvideo();`

Nachfolgende Textausgaben erfolgen mit hoher Intensität.

insline `void insline();`

Fügt eine leere Zeile ein.

lowvideo `void lowvideo();`

Nachfolgende Textausgaben erfolgen mit niedriger Intensität.

normvideo `void normvideo();`

Nachfolgende Textausgaben erfolgen mit normaler Intensität.

overflow `virtual int overflow(int = EOF);`

Das *conbuf*-Objekt wird in die Zielausgabe geschrieben.

setcursortype `void setcursortype(int cur_type);`

Auswahl der Cursor-Form.

textattr `void textattr(int newattribute);`

Auswahl der Text-Attribute.

textbackgroundcolor `void textbackgroundcolor(int newcolor);`

Auswahl der Texthintergrundfarbe.

textcolor `void textcolor( int newcolor);`

Auswahl der Zeichenfarbe im Textmodus.

textmode `static void textmode(int newmode);`

Schaltet den Bildschirm in den Textmodus.

wherex
```
int wherex();
```
Liefert die horizontale Cursor-Position zurück.

wherey
```
int wherey();
```
Liefert die vertikale Cursor-Position zurück.

window
```
void window(int left, int top, int right, int bottom);
```
Definiert das aktive Textfenster.

Klasse constream — constrea.h

constream ist nur für Textbildschirmanwendungen verfügbar.

Stellt Ausgabe-Streams für Textbildschirmanwendungen zur Verfügung.

Public-Konstruktoren

```
constream();
```
Erzeugt einen neuen, nicht verbundenen Ausgabe-Stream für den Textbildschirm.

Public-Elementfunktionen

clrscr
```
void clrscr();
```
Löscht den Bildschirm.

rdbuf
```
conbuf *rdbuf();
```
Liefert einen Zeiger auf das *conbuf*-Objekt, das zu diesem *constream*-Objekt gehört.

textmode
```
void textmode(int newmode);
```
Schaltet den Bildschirm in den Textmodus.

window
```
void window(int left, int top, int right, int bottom);
```
Definiert das aktive Textfenster.

Klasse filebuf — fstream.h

Diese Klasse spezialisiert die Klasse *streambuf* und verwendet Dateien für die Ein-/Ausgabe von Zeichen. Die Klasse *filebuf* verwaltet die Reservierung und Freigabe von Puffern sowie Suchoperationen innerhalb der Da-

tei. Ungepufferte Ein-/Ausgabe ist bei Verwendung des entsprechenden Konstruktors oder der Elementfunktion *filebuf::setbuf* ebenfalls möglich. Dateien werden standardmäßig im Modus *openprot* geöffnet, der Lese- und Schreibzugriffe erlaubt. Auf Seite 345 finden Sie eine Liste der Dateiöffnungsmodi.

Die Klasse *filebuf* enthält lediglich die Grundoperationen für die Datei-Ein-/Ausgabe. Die Ein-/Ausgabe in ein *filebuf*-Objekt kann nur mit den Grundfunktionen von *streambuf* erfolgen. Formatierungsmöglichkeiten werden in den spezialisierteren Klassen angeboten.

Public-Konstruktoren

```
filebuf();
```

Erzeugt ein *filebuf*-Objekt, das keiner Datei zugeordnet ist.

```
filebuf(int fd);
```

Erzeugt ein *filebuf*-Objekt, das der Datei mit dem Dateideskriptor *fd* zugeordnet ist.

```
filebuf(int fd, char *buf, int n);
```

Erzeugt ein *filebuf*-Objekt, das der Datei mit dem Deskriptor *fd* zugeordnet ist. Dabei wird *buf* als Speicherbereich verwendet. Die Größe von *buf* muß zur Speicherung von *n* Bytes ausreichen. Ist *buf* gleich NULL oder *n* kein positiver Wert, dann wird ein ungepuffertes *filebuf*-Objekt erzeugt.

Public-Datenelemente

openprot

```
static const int openprot;
```

Der Standard-Dateischutz. Der genaue Wert von *openprot* ist für den Benutzer nicht interessant. Die Aufgabe von *openprot* besteht darin, die Lese- und Schreibberechtigungen der Datei festzulegen.

Public-Elementfunktionen

attach

```
filebuf* attach(int fd);
```

Verbindet das geschlossene *filebuf*-Objekt mit der durch den Deskriptor *fd* angegebenen Datei. Wurde der Puffer bereits geöffnet, dann wird *attach* nicht erfolgreich ausgeführt und liefert NULL zurück. Andernfalls wird der Dateipuffer mit *fd* verbunden.

close

```
filebuf* close();
```

Schreibt den Puffer in die Datei und schließt die Datei. Normalerweise muß die Funktion *close* bei Programmende nicht explizit aufgerufen werden, da das ordnungsgemäße Schließen der Datei durch den *filebuf*-Destruktor automatisch gewährleistet wird. Ein expliziter Aufruf ist dann sinnvoll, wenn Sie die Verbindung des *filebuf*-Objekts mit der Datei innerhalb Ihres Programms trennen möchten.

Im Fehlerfall, beispielsweise bei bereits geschlossener Datei, wird der Wert 0 zurückgeliefert. Andernfalls wird eine Referenz auf das *filebuf*-Objekt zurückgeliefert (der **this-**Zeiger).

fd

```
int fd()
```

Liefert den Dateideskriptor oder EOF zurück.

is_rtl_open

```
int is_rtl_open()
```

Liefert bei geöffneter Datei einen Wert ungleich null zurück.

open

```
filebuf* open(const char *filename, int mode, int prot = filebuf::openprot);
```

Öffnet die durch *filename* angegebene Datei und stellt die Verbindung zu ihr her. Die Datei wird in dem durch den Parameter *mode* angegebenen Modus geöffnet.

overflow

```
virtual int overflow(int c = EOF);
```

Der Puffer wird in die Zieldatei geschrieben. Jede abgeleitete Klasse sollte die hierfür erforderlichen Maßnahmen implementieren.

seekoff

```
virtual streampos seekoff(streamoff offset, dir ios::seek_dir, int mode);
```

Verschiebt den Ein-/Ausgabezeiger um *offset* Bytes. Der Dateizeiger wird dabei relativ zu seiner aktuellen Position in die durch den Parameter *dir* angegebene Richtung verschoben. Der Parameter *mode* kann die Werte *ios::in* (Eingabe), *ios::out* (Ausgabe) oder auch beides annehmen. Bei *ios::in* wird der Ausgabezeiger, bei *ios::out* wird der Ausgabezeiger verschoben.

Bei erfolgreicher Ausführung liefert die Funktion *seekoff* einen Wert des Typs *streampos* zurück, der die neue Position des Dateizeigers wiedergibt.

Ein Fehler kann auftreten, wenn die Datei keine Verschiebung des Dateizeigers unterstützt oder wenn Sie eine ungültige Verschiebung – beispielsweise hinter das Dateiende – anfordern. Im Fehlerfall liefert *seekoff* EOF zurück und die Position des Dateizeigers ist undefiniert.

setbuf

```
virtual streambuf* setbuf(char *buf, int len);
```

Reserviert für das *filebuf*-Objekt einen durch *buf* angegebenen Puffer der Größe *len* Bytes. Ist *buf* gleich NULL oder *len* kein positiver Wert, dann ist das *filebuf*-Objekt ungepuffert.

Bei erfolgreicher Ausführung liefert *setbuf* einen Zeiger auf das *filebuf*-Objekt zurück. Ein Fehler tritt auf, wenn die Datei offen und ein Puffer bereits reserviert worden ist. Im Fehlerfall liefert *setbuf* NULL zurück und ändert den Pufferstatus nicht.

sync

```
virtual int sync();
```

Sorgt dafür, daß die internen Datenstrukturen mit der externen Stream-Repräsentation übereinstimmen.

underflow

```
virtual int underflow();
```

Eingaben werden verfügbar gemacht. Diese Elementfunktion wird aufgerufen, wenn sich im Eingabepuffer keine weiteren Daten befinden. Jede abgeleitete Klasse sollte die erforderlichen Aktionen definieren.

Klasse fstream — fstream.h

Diese von *fstreambase* und *iostream* abgeleitete Stream-Klasse realisiert gleichzeitige Ein-/Ausgaben für *filebuf*-Objekte.

Public-Konstruktoren

```
fstream();
```

Erzeugt ein *fstream*-Objekt, ohne es mit einer Datei zu verbinden.

```
fstream(const char *name, int mode, int prot = filebuf::openprot);
```

Erzeugt ein *fstream*-Objekt, öffnet eine Datei mit den durch *mode* angegebenen Zugriffsrechten und verbindet beide. Auf Seite 345 finden Sie eine Liste der von *ios::open_mode* unterstützten Dateiöffnungsmodi

```
fstream(int fd);
```

Erzeugt ein *fstream*-Objekt und verbindet es mit der durch den Deskriptor *fd* angegebenen offenen Datei.

```
fstream(int fd, char *buf, int n);
```

Erzeugt ein *fstream*-Objekt, das mit der durch den Dateideskriptor *fd* angegebenen Datei verbunden wird, und benutzt *buf* als Speicherbereich. Die Größe von *buf* reicht zur Speicherung von *n* Bytes aus. Ist *buf* gleich

NULL oder *n* ein nicht positiver Wert, dann ist das *fstream*-Objekt ungepuffert.

Public-Elementfunktionen

open

```
void open(const char *name, int mode, int prot = filebuf::openprot);
```

Öffnet die durch *name* angegebene Datei für ein *fstream*-Objekt. Der Öffnungsmodus wird durch den Parameter *mode* bestimmt.

rdbuf

```
filebuf* rdbuf();
```

Liefert das verwendete *filebuf*-Objekt zurück.

Klasse fstreambase fstream.h

Diese von *ios* abgeleitete Stream-Klasse enthält allgemeine Operationen für Datei-Streams. *fstreambase* ist Basisklasse für *fstream*, *ifstream* und *ofstream*.

Public-Konstruktoren

```
fstreambase();
```

Erzeugt ein *fstreambase*-Objekt, ohne es mit einer Datei zu verbinden.

```
fstreambase(const char *name, int mode, int = filebuf::openprot);
```

Erzeugt ein *fstreambase*-Objekt, öffnet die durch *name* angegebene Datei mit den durch *mode* angegebenen Zugriffsrechten und verbindet beide.

```
fstreambase(int fd);
```

Erzeugt ein *fstreambase*-Objekt und verbindet es mit der durch den Deskriptor *fd* angegebenen offenen Datei.

```
fstreambase(int fd, char *buf, int len);
```

Erzeugt ein *fstreambase*-Objekt, das mit der durch den Dateideskriptor *fd* angegebenen Datei verbunden wird, und benutzt *buf* als Puffer der Größe *len*.

Public-Elementfunktionen

attach

```
void attach(int fd);
```

Stellt die Verbindung zu der durch *fd* angegebenen offenen Datei her.

close `void close();`

Schließt das *filebuf*-Objekt und die zugehörige Datei.

open `void open(const char *name, int mode, int prot = filebuf::openprot);`

Öffnet eine Datei für ein *fstreambase*-Objekt. Der Öffnungsmodus wird durch den Parameter *mode* festgelegt.

rdbuf `filebuf* rdbuf();`

Liefert das verwendete *filebuf*-Objekt zurück.

setbuf `void setbuf(char *buf, int len);`

Reserviert einen Speicherbereich, auf den *buf* zeigt. Der Speicherbereich ist groß genug, um *len* Bytes aufzunehmen.

Klasse ifstream fstream.h

Diese von *fstreambase* und *istream* abgeleitete Stream-Klasse implementiert die Eingabeoperationen für ein *filebuf*-Objekt.

Public-Konstruktoren

```
ifstream();
```

Erzeugt ein *ifstream*-Objekt, ohne es mit einer Datei zu verbinden.

```
ifstream(const char *name, int mode = ios::in, int prot = filebuf::open-
prot);
```

Erzeugt ein *ifstream*-Objekt, öffnet die durch *name* angegebene Datei für geschützte Lesezugriffe, und verbindet beide. Existiert die Datei nicht, so wird sie standardmäßig nicht erzeugt.

```
ifstream(int fd);
```

Erzeugt ein *ifstream*-Objekt und verbindet es mit der durch den Deskriptor *fd* angegebenen offenen Datei.

```
ifstream(int fd, char *buf, int buf_len);
```

Erzeugt ein *ifstream*-Objekt, das mit der durch den Dateideskriptor *fd* angegebenen Datei verbunden wird, und benutzt *buf* als Puffer der Größe *len*.

Public-Elementfunktionen

open

```
void open(const char *name, int mode, int prot = filebuf::openprot);
```

Öffnet eine Datei für ein *ifstream*-Objekt.

rdbuf

```
filebuf* rdbuf();
```

Liefert das verwendete *filebuf*-Objekt zurück.

Klasse ios — iostream.h

Diese Klasse enthält allgemeine Ein-/Ausgabeoperationen. Die abgeleiteten Klassen (*istream*, *ostream*, *iosteam*) spezialisieren diese und bieten formatierte Ein-/Ausgabeoperationen. Die Klasse *ios* ist Basisklasse für *istream*, *ostream*, *fstreamable* und *strstreambase*.

Public-Datenelemente

Die folgenden drei Konstanten werden für den zweiten Parameter der Funktion *setf* benötigt:

```
static const long  adjustfield; // left | right | internal
static const long  basefield;   // dec | oct | hex
static const long  floatfield;  // scientific | fixed
```

Stream-Suchrichtung:

```
enum seek_dir { beg=0, cur=1, end=2 };
```

Modus der Stream-Operationen. Sie können logisch ODER-verknüpft werden:

```
enum open_mode  {
    app,        Daten anhängen – immer am Dateiende schreiben
    ate,        Nach Öffnen auf Dateiende positionieren
    in,         Öffnen für Eingabe (Standard für ifstream)
    out,        Öffnen für Ausgabe (Standard für ofstream)
    binary,     Öffnen im Binärmodus
    trunc,      Dateiinhalt löschen, falls Datei existiert (Standard,
                falls out und weder ate noch app angegeben wurde)
    nocreate,   open erzeugt Fehler, falls Datei nicht existiert
    noreplace,  Wenn Datei existiert, dann führt das Öffnen für Ausgabe-
                operationen ohne die Angabe von ate oder app zu Fehlern
};
```

Format-Flags, die in den Elementfunktionen *flags*, *setf* und *unsetf* verwendet werden:

```
enum {
   skipws,      Whitespace-Zeichen bei der Eingabe überspringen
   left,        Linksbündige Ausgabe
   right,       Rechtsbündige Ausgabe
   internal,    Freiraum nach Vorzeichen oder Basis
   dec,         Dezimalkonvertierung
   oct,         Oktalkonvertierung
   hex,         Hexadezimalkonvertierung
   showbase,    Basis bei Ausgabe darstellen
   showpoint,   Dezimalpunkt bei Gleitkommaausgaben darstellen
   uppercase,   Hex-Ausgabe in Großbuchstaben
   showpos,     '+' Zeichen bei positiven Integerwerten darstellen
   scientific,  Gleitkommawerte in Exponential-Schreibweise (E)
   fixed,       Feste Stellenanzahl bei Gleitkommawerten verwenden
   unitbuf,     Alle Streams nach Einfügeoperation ausgeben
   stdio,       stdout und stderr nach Einfügeoperation ausgeben
};
```

Protected-Datenelemente

```
streambuf   *bp;           // Der zugehörige streambuf
int         x_fill;        // Füllzeichen für Ausgabe
long        x_flags;       // Format-Flags
int         x_precision;   // Genauigkeit der Fließkomma-Ausgaben
int         state;         // Aktueller Status von streambuf
ostream     *x_tie;        // Der zugehörige ostream, sofern er existiert
int         x_width;       // Feldbreite der Ausgabe
```

Public-Konstruktoren

```
ios(streambuf *);
```

Verbindet das angegebene *streambuf*-Objekt mit dem Stream.

Protected-Konstruktoren

```
ios();
```

Erzeugt ein *ios*-Objekt ohne zugehöriges *streambuf*-Objekt.

Public-Elementfunktionen

bad

```
int bad();
```

Liefert einen Wert ungleich null zurück, wenn ein Fehler aufgetreten ist.

bitalloc

```
static long bitalloc();
```

Anforderung eines neuen Flag-Bit-Satzes. Der Rückgabewert kann zum Setzen, Löschen und Testen der benutzerdefinierten Format-Flags verwendet werden.

clear

```
void clear(int = 0);
```

Setzt den Stream-Status auf den angegebenen Wert.

eof

```
int eof();
```

Liefert bei erreichtem Dateiende einen Wert ungleich null zurück.

fail

```
int fail();
```

Liefert bei einer fehlgeschlagenen Operation einen Wert ungleich null zurück.

fill

```
char fill();
```

Liefert das aktuelle Füllzeichen zurück.

fill

```
char fill(char);
```

Legt das Füllzeichen neu fest und lieferte das bisherige Zeichen zurück.

flags

```
long flags();
```

Liefert die aktuellen Format-Flags zurück.

flags

```
long flags(long);
```

Setzt die Format-Flags auf den angegebenen **long**-Wert und liefert die bisherigen Flags zurück. Mit *flags(0)* werden die Standard-Flags wiederherstellt.

good

```
int good();
```

Liefert einen Wert ungleich null zurück, wenn keine Status-Bits gesetzt sind.

precision

```
int precision();
```

Liefert die aktuelle Genauigkeit der Gleitkommadarstellung zurück.

precision `int precision(int);`

Setzt die Gleitkommagenauigkeit und liefert die bisherige Einstellung zurück.

rdbuf `streambuf* rdbuf();`

Liefert einen Zeiger auf das zugehörige *streambuf*-Objekt zurück.

rdstate `int rdstate();`

Liefet den Status des Stream zurück.

setf `long setf(long);`

Setzt die Flags auf die Bits des angegebenen **long-**Wert und liefert die bisherigen Flags zurück.

setf `long setf(long _setbits, long _field);`

Die Bits, die den gesetzten Bits des Parameters *_field* entsprechen, werden gelöscht und anschließend entsprechend der Bits in *_setbits* neu gesetzt.

sync_with_stdio `static void sync_with_stdio();`

Verbindet stdio-Dateien und *iostream*-Objekte. Diese Funktion sollte in neuen Programm nicht verwendet werden.

tie `ostream* tie();`

Liefert den »verbundenen Stream« (tied stream) oder den Wert NULL, wenn keiner existiert. Verbundene Streams sind Streams, die so miteinander verknüpft sind, daß bei der Benutzung eines Stream auch der andere angesprochen wird. *cin* und *cout* sind beispielsweise miteinander verbunden, da vor jeder Benutzung von *cin* zuerst *cout* ausgegeben wird.

tie `ostream* tie(ostream *out);`

Verbindet einen anderen Stream mit dem Ausgabe-Stream *out* und liefert den bisher verbundenen Stream zurück.

Muß ein Eingabe-Stream Zeichen übernehmen oder benötigt ein Ausgabe-Stream weitere Zeichen, dann wird der verbundene Stream zuvor automatisch geschrieben. Standardmäßig sind *cin*, *cerr* und *clog* mit *cout* verbunden.

unsetf `long unsetf(long f);`

Löscht die dem Parameter *f* entsprechenden Bits und liefert einen **long-**Wert zurück, der die bisherige Einstellung repräsentiert.

width `int width();`

Liefert die aktuelle Breiteneinstellung zurück.

width `int width(int);`

Setzt die Breiteneinstellung und liefert den bisherigen Wert zurück.

xalloc `static int xalloc();`

Liefert einen Array-Index auf ein bisher nicht benutztes Speicherwort, das für benutzerdefinierte Format-Flags verwendet werden kann.

Protected-Elementfunktionen

init `void init(streambuf *);`

Führt die eigentliche Initialisierung durch.

setstate `void setstate(int);`

Setzt alle Statusbits.

Klasse iostream — iostream.h

Diese von *istream* und *ostream* abgeleitete Klasse ist eine Mischung ihrer Basisklassen, die sowohl Eingabe als auch Ausgabe für Streams zuläßt. Sie dient als Basisklasse für *fstream* und *strstream*.

Public-Konstruktoren

`iostream(streambuf *);`

Verbindet das angegebene *streambuf*-Objekt mit dem Stream.

Klasse iostream_withassign — fstream.h

Diese Klasse implementiert *iostream*-Objekte, die zusätzlich über einen Zuweisungsoperator verfügen.

Public-Konstruktoren

`iostream_withassign();`

Standardkonstruktor (ruft den Konstruktor von *iostream* auf).

Public-Elementfunktionen

Keine (obwohl der Zuweisungsoperator überladen ist).

Klasse istream iostream.h

Diese Klasse stellt formatierte und unformatierte Eingabeoperationen aus einer *streambuf*-Klasse zur Verfügung. Der Operator **>>** ist für alle vordefinierte Typen überladen. Die Klasse *istream* ist Basisklasse für *ifstream*, *iostream*, *istrstream* und *istream_withassign*.

Public-Konstruktoren

```
istream(streambuf *);
```

Verbindet das angegebene *streambuf*-Objekt mit dem Stream.

Public-Elementfunktionen

gcount

```
int gcount();
```

Liefert die Anzahl der zuletzt gelesenen Zeichen.

get

```
int get();
```

Liest das nächste Zeichen oder EOF.

get

```
istream& get(char *buf, int len, char delim = '\n');
istream& get(signed char *buf, int len, char delim = '\n');
istream& get(unsigned char *buf, int len, char delim = '\n');
```

Liest Zeichen und speichert sie im durch *buf* angegebenen Puffer, bis das durch *delim* angegebene Begrenzungszeichen oder EOF auftritt, bzw. bis (*len*-1) Zeichen gelesen wurden. Ein abschließendes Nullzeichen wird stets in den Ausgabe-String geschrieben, das Begrenzungszeichen jedoch nicht. Es tritt nur dann ein Fehler auf, wenn keine Zeichen gelesen werden konnten.

Die Funktion *get* endet fehlerhaft, wenn EOF auftritt, bevor Zeichen gespeichert wurden. Im Fehlerfall setzt *get* das Bit *ios::failbit*.

get

```
istream& get(char &ch);
istream& get(signed char &ch);
istream& get(unsigned char &ch);
```

Liest ein einzelnes Zeichen in die Referenzvariable *ch*.

get

```
istream& get(streambuf &sbuf, char delim = '\n');
```

Liest solange Zeichen in den angegebenen Parameter *sbuf*, bis das Zeichen *delim* auftritt.

getline

```
istream& getline(char *buf, int len, char);
istream& getline(signed char *buf, int len, char delim = '\n');
istream& getline(unsigned char *buf, int len, char delim = '\n');
```

Wie *get*, außer, daß das Begrenzungszeichen ebenfalls gelesen wird. Normalerweise wird *delim* nicht in *buf* kopiert. Tritt das Begrenzungszeichen jedoch genau dann auf, wenn *len* Zeichen gelesen wurden, dann wird *delim* nicht mehr gelesen.

ignore

```
istream& ignore(int n = 1, int delim = EOF);
```

Bewirkt, daß bis zu *n* Zeichen im Eingabe-Stream übersprungen werden. Beim Auftreten des Zeichen *delim* bricht die Funktion ab.

ipfx

```
istream& ipfx(int n = 0);
```

Die Funktion *ipfx* wird von Eingabefunktionen vor dem Zugriff auf einen Eingabe-Stream aufgerufen. Funktionen mit formatierter Ausgabe rufen *ipfx(0)* auf; bei unformatierter Eingabe wird *ipfx(1)* aufgerufen.

peek

```
int peek();
```

Liefert des nächste Zeichen zurück, ohne es aus dem Stream auszulesen.

putback

```
istream& putback(char);
```

Schreibt ein Zeichen in den Stream zurück.

read

```
istream& read(char*, int);
istream& read(signed char*, int);
istream& read(unsigned char*, int);
```

Liest eine gegebene Anzahl von Zeichen aus dem Stream in ein Array. Im Fehlerfall können Sie mit *gcount* die Anzahl der tatsächlich gelesenen Zeichen ermitteln.

seekg

```
istream& seekg(streampos pos);
```

Positioniert auf eine absolute Position im Eingabe-Stream.

seekg

```
istream& seekg(streamoff offset, seek_dir dir);
```

Positioniert auf die Position, die *offset* Bytes von der aktuellen Position des Eingabe-Stream entfernt ist. Der Parameter *dir* gibt dabei die Richtung des Versatzes an und ist wie folgt definiert: **enum** *seek_dir {beg, cur, end };*

Für die Positionierung in einem Ausgabe-Stream verwenden Sie *ostream::seekp.*

Für die Positionierung in einem Eingabe-Puffer verwenden Sie *seekpos* oder *seekoff*.

tellg

```
streampos tellg();
```

Liefert die aktuelle Stream-Position zurück. Im Fehlerfall wird ein negativer Wert zurückgeliefert.

Benutzen Sie *ostream::tellp*, um die Position in einem Ausgabe-Stream zu ermitteln.

Protected-Elementfunktionen

eatwhite

```
void eatwhite();
```

Liest aufeinander folgende Whitespace-Zeichen.

Klasse istream_withassign — iostream.h

Diese Klasse implementiert *istream*-Objekte mit zusätzlichem Zuweisungsoperator.

Public-Konstruktoren

```
istream_withassign();
```

Standardkonstruktor (ruft den Konstruktor von *istream* auf).

Public-Elementfunktionen

Keine (obwohl der Zuweisungsoperator überladen ist).

Klasse istrstream — strstrea.h

Stellt Eingabeoperationen für die Klasse *strstreambuf* zur Verfügung. *istrstream* ist von *strstreambase* und *istream* abgeleitet.

Public-Konstruktoren

```
istrstream(char *);
istrstream(signed char *);
istrstream(unsigned char *);
```

Jeder der obigen Konstruktoren erzeugt ein *istrstream*-Objekt aus einem angegeben String (ein Nullzeichen wird niemals ausgelesen). Im Kapitel 1 des *Programmierhandbuch* finden Sie eine Erläuterung zu den drei Zeichentypen.

```
istrstream(char *str, int n);
istrsteam(signed char *str, int);
istrstream(unsigned char *str, int);
```

Jeder der obigen Konstruktoren erzeugt ein *istrstream*-Objekt unter Verwendung von maximal *n* Bytes von *str*. Im Kapitel 1 des *Programmierhandbuchs finden Sie eine Erläuterung zu den drei Zeichentypen.*

Klasse ofstream — fstream.h

Diese Klasse stellt Eingabeoperationen für ein *filebuf*-Objekt zur Verfügung und ist von *fstreambase* und *ostream* abgeleitet.

Public-Konstruktoren

```
ofstream();
```

Erzeugt einen *ofstream*-Objekt, das mit keiner Datei verbunden ist.

```
ofstream(const char *name, int mode = ios::out, int prot = filebuf::openprot);
```

Erzeugt einen *ofstream*-Objekt, öffnet eine Datei und verbindet beide.

```
ofstream(int fd);
```

Erzeugt ein *ofstream*-Objekt und verbindet es mit der durch den Deskriptor *fd* angegebenen offenen Datei.

```
ofstream(int fd, char *buf, int len);
```

Erzeugt ein *ofstream*-Objekt, das mit der durch den Dateideskriptor *fd* angegebenen Datei verbunden wird, und benutzt *buf* als Puffer der Größe *len*.

Public-Elementfunktionen

open

```
void open(const char *name, int mode = ios::out, int prot = filebuf::open-
prot);
```

Öffnet eine Datei für ein *ofstream*-Objekt.

rdbuf

```
filebuf* rdbuf();
```

Liefert das verwendete *filebuf*-Objekt zurück.

Klasse ostream iostream.h

Die Klasse *ostream* stellt formatierte und unformatierte Ausgabeoperationen für *streambuf*-Objekte bereit. Der Operator **<<** ist für alle vordefinierten Typen überladen. Diese von *ios* abgeleitete Klasse ist Basisklasse für *constream*, *iostream*, *ofstream*, *ostrstream* und *ostrstream_withassign*.

Public-Konstruktoren

```
ostream(streambuf *);
```

Verbindet das angegebene *streambuf*-Objekt mit dem Stream.

Public-Elementfunktionen

flush

```
ostream& flush();
```

Leert den Stream.

opfx

```
int opfx();
```

Die Funktion *opfx* wird von Ausgabefunktionen vor dem Zugriff auf einen Ausgabe-Stream aufgerufen. *opfx* liefert 0 zurück, wenn das *ostream*-Objekt einen Fehlerstatus ungleich null hat. Andernfalls wird ein Wert ungleich null zurückgeliefert.

osfx

```
void osfx();
```

Die Funktion *osfx* führt Operationen nach Ausgaben durch. Ist *ios::unitbuf* eingeschaltet, dann leert *opfx* den Stream. Im Fehlerfall setzt die Funktion das Bit *ios::failbit*.

put

```
ostream& put(unsigned char ch);
ostream& put(char ch);
ostream& put(signed char ch);
```

Fügt das angegebene Zeichen in den Stream ein.

seekp

```
ostream& seekp(streampos);
```

Positioniert auf eine absolute Position im Stream (wie sie von *tellg* zurückgeliefert wird).

seekp

```
ostream& seekp(streamoff, seek_dir);
```

Positioniert auf die Position, die *streamoff* Bytes von der aktuellen Position des Eingabe-Stream entfernt ist. Der Parameter *dir* gibt dabei die Richtung des Versatzes an und ist wie folgt definiert: **enum** *seek_dir {beg, cur, end };*

tellp

```
streampos tellp();
```

Liefert die aktuelle Stream-Position zurück.

write

```
ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);
ostream& write(const char*, int n);
```

Fügt *n* Zeichen (auch Nullzeichen) in den Stream ein.

Klasse ostream_withassign — iostream.h

Diese Klasse implementiert *ostream*-Objekte, die zusätzlich über einen Zuweisungsoperator verfügen.

Public-Konstruktoren

```
ostream_withassign();
```

Standardkonstruktor (ruft den Konstruktor von *ostream* auf).

Public-Elementfunktionen

Keine (obwohl der Zuweisungsoperator überladen ist).

Klasse ostrstream — strstrea.h

Diese Klasse stellt die Ausgabeoperationen für *strstreambuf* zur Verfügung und ist von *strstreambase* und *ostream* abgeleitet.

Public-Konstruktoren

```
ostrstream();
```

Erzeugt ein dynamisches *ostrstream*-Objekt.

```
ostrstream(char *buf, int len, int mode = ios::out);
ostrstream(signed char *buf, int len, int mode = ios::out);
ostrstream(unsigned char *buf, int len, int mode = ios::out);
```

Jeder der obigen Konstruktoren erzeugt ein *istrstream*-Objekt unter Verwendung des angegebenen Puffers der Größe *n* Bytes. Wenn *mode* gleich *ios:app* oder *ios::ate* ist, dann wird der Ein-/Ausgabezeiger auf das Nullzeichen des Strings positioniert. Im Kapitel 1 des *Programmierhandbuch* finden Sie eine Erläuterung zu den drei Zeichentypen.

Public-Elementfunktionen

pcount

`int pcount();`

Liefert die Anzahl der aktuell im Puffer gespeicherten Bytes zurück.

str

`char *str();`

Liefert den Puffer zurück und verhindert weitere Pufferänderungen. Wenn der Puffer dynamisch war, dann müssen Sie ihn freigeben.

Klasse streambuf — iostream.h

Diese Klasse ist Basisklasse für alle anderen gepufferten Klassen. Sie stellt eine Pufferschnittstelle zwischen Ihren Daten und Speicherbereichen wie Hauptspeicher oder Geräten zur Verfügung. Die von *streambuf* erzeugten Puffer werden als Eingabe-, Ausgabe- und Reservebereiche bezeichnet. Ihr Inhalt wird über **char-**Zeiger angesprochen und verändert.

Die von *streambuf* durchgeführte Pufferung ist relativ einfach. Normalerweise greifen Anwendungen über einen von *ios* gesetzten *streambuf*-Zeiger auf die Puffer und Pufferfunktionen zu. Die Klasse *ios* stellt einen Zeiger auf *streambuf* bereit, der den höherentwickelten Klassen einen transparenten Zugriff auf die Pufferroutinen erlaubt. Die höherentwickelten Klassen stellen eine formatierte Ein-/Ausgabe zur Verfügung.

Public-Konstruktoren

`streambuf();`

Erzeugt ein leeres Pufferobjekt.

`streambuf(char *buf, int size);`

Erzeugt einen leeren Puffer *buf* und richtet einen Reservebereich der Größe *size* Bytes ein.

Public-Elementfunktionen

in_avail

`int in_avail();`

Liefert die Anzahl der Zeichen im Eingabepuffer zurück.

out_waiting

`int out_waiting();`

Liefert die Anzahl der im Ausgabepuffer verbleibenden Zeichen zurück.

sbumpc

```
int sbumpc();
```

Liefert das aktuelle Zeichen aus dem Eingabepuffer und positioniert auf das nächste Zeichen.

seekoff

```
virtual streampos seekoff(streamoff, ios::seek_dir, int = (ios::in |
ios::out);
```

Verschiebt den Ein-/Ausgabezeiger (das dritte Argument legt den zu verwenden Zeiger fest) relativ zur aktuellen Position.

seekpos

```
virtual streampos seekpos(streampos, int = (ios::in | ios::out));
```

Verschiebt den Ein-/Ausgabezeiger auf eine absolute Position.

setbuf

```
virtual streambuf* setbuf(char *, int);
```

Stellt eine Verbindung zu dem angegebenen Puffer her.

sgetc

```
int sgetc();
```

Liefert das nächste Zeichen im Eingabepuffer, ohne es auszulesen.

sgetn

```
int sgetn(char*, int n);
```

Liest die nächsten *n* Zeichen aus dem Eingabepuffer.

snextc

```
int snextc();
```

Positioniert auf das nächste Zeichen im Eingabepuffer und liefert es zurück.

sputbackc

```
int sputbackc(char);
```

Schreibt ein Zeichen in den Eingabepuffer zurück.

sputc

```
int sputc(int);
```

Schreibt ein Zeichen in den Ausgabepuffer.

sputn

```
int sputn(const char*, int n);
```

Schreibt *n* Zeichen in den Ausgabepuffer.

stossc

```
void stossc();
```

Positioniert auf das nächste Zeichen im Eingabepuffer.

Protected-Elementfunktionen

allocate

```
int allocate();
```

Richtet einen Pufferbereich ein.

base `char *base();`

Liefert die Startadresse des Pufferbereichs zurück.

blen `int blen();`

Liefert die Länge des Pufferbereichs zurück.

eback `char *eback();`

Liefert die Größe des Rückgabebereichs im Eingabepuffer zurück.

ebuf `char *ebuf();`

Liefert die um eins erhöhte Endadresse des Pufferbereichs.

egptr `char *egptr();`

Liefert die um eins erhöhte Endadresse des Eingabebereichs.

epptr `char *epptr();`

Liefert die um eins erhöhte Endadresse des Ausgabebereichs.

gbump `void gbump(int);`

Bewegt den Eingabezeiger weiter.

gptr `char *gptr();`

Liefert die nächste Position im Eingabebereich zurück.

pbase `char *pbase();`

Liefert die Startadresse des Ausgabebereichs.

pbump `void pbump(int);`

Bewegt den Ausgabezeiger weiter.

pptr `char *pptr();`

Liefert die nächste Position im Ausgabebereich zurück.

setb `void setb(char *, char *, int = 0 );`

Legt den Pufferbereich fest.

setg `void setg(char *, char *, char *);`

Initialisiert die Eingabezeiger.

setp `void setp(char *, char *);`

Initialisiert die Ausgabezeiger.

unbuffered `void unbuffered(int);`

Setzt den Pufferstatus.

unbuffered

```
int unbuffered();
```

Liefert einen Wert ungleich null, wenn die Pufferung abgeschaltet ist.

Klasse strstreambase strstrea.h

Diese Klasse spezialisiert *ios* für String-Streams und besteht mit Ausnahme von *strstreambase::rdbuf* nur aus protected-Elementen. Die Klasse *strstreambase* ist Basisklasse für *strstream*, *istrstream* und *ostrstream*.

Public-Konstruktoren

```
strstreambase();
```

Erzeugt ein leeres *strstreambase*-Objekt.

```
strstreambase(char *, int, char *start);
```

Erzeugt ein leeres *strstreambase*-Objekt mit dem angegebenen Puffer und der angegebenen Startposition.

Public-Elementfunktionen

rdbuf

```
strstreambuf * rdbuf();
```

Liefert einen Zeiger auf das zugehörende *strstreambuf*-Objekt.

Klasse strstreambuf strstrea.h

Spezialisiert die Klasse *streambuf* für Formatierung im Speicher.

Public-Konstruktoren

```
strstreambuf();
```

Erzeugt ein dynamisches *strstreambuf*-Objekt. Speicher wird dynamisch reserviert, wenn er benötigt wird.

```
strstreambuf(void * (*)(long), void (*)(void *));
```

Erzeugt einen dynamischen Puffer mit den angegebenen Funktionen zur Reservierung und Freigabe von Speicher.

```
strstreambuf(int n);
```

Erzeugt ein dynamisches *strstreambuf*-Objekt, für das anfangs ein Puffer der Mindestgröße *n* Bytes reserviert wird.

```
strstreambuf(char*, int, char *strt = 0);
strstreambuf(signed char *, int, signed char *strt = 0);
strstreambuf(unsigned char *, int, unsigned char *strt = 0);
```

Jeder der drei obigen Konstruktoren erzeugt ein statisches *strstreambuf*-Objekt mit dem angegebenen Puffer. Ist der Parameter *strt* ungleich null, so begrenzt er den Puffer. Im Kapitel 1 des *Programmierhandbuch* finden Sie eine Erläuterung zu den drei Zeichentypen.

Public-Elementfunktionen

doallocate

```
virtual int doallocate();
```

Führt Low-Level-Speicherreservierungen für Puffer durch.

freeze

```
void freeze(int = 1);
```

Wenn der Eingabeparameter ungleich null ist, wird das Speichern von Zeichen im Puffer verhindert. Durch Übergabe des Werts 0 kann das Speichern wieder zugelassen werden.

overflow

```
virtual int overflow(int);
```

Leert den Puffer, indem er in sein Ziel geschrieben wird. Jede abgeleitete Klasse sollte die hierfür erforderlichen Aktionen definieren.

seekoff

```
virtual streampos seekoff(streamoff, ios::seek_dir, int);
```

Verschiebt den Zeiger relativ zur aktuellen Position.

setbuf

```
virtual streambuf* setbuf(char*, int);
```

Bestimmt den zu verwendenden Puffer.

str

```
char *str();
```

Liefert einen Zeiger auf den Puffer zurück und verhindert weitere Pufferänderungen.

sync

```
virtual int sync();
```

Sorgt dafür, daß die internen Datenstrukturen mit der externen Stream-Repräsentation übereinstimmen.

underflow

```
virtual int underflow();
```

Eingaben werden verfügbar gemacht. Diese Elementfunktion wird aufgerufen, wenn ein Zeichen angefordert wird und sich im Eingabepuffer keine weiteren Daten befinden. Jede abgeleitete Klasse sollte die erforderlichen Aktionen definieren.

Klasse strstream — strstrea.h

Stellt gleichzeitige Ein- und Ausgabeoperationen für *strstreambuf*-Objekte zur Verfügung. Diese Klasse ist von *strstreambase* und *iostream* abgeleitet.

Public-Konstruktoren

```
strstream();
```

Erzeugt ein dynamisches *strstream*-Objekt.

```
strstream(char *buf, int sz, int mode);
strstream(signed char *buf, int sz, int mode);
strstream(unsigned char *buf, int sz, int mode);
```

Jeder der obigen Konstruktoren erzeugt ein *strstream*-Objekt unter Verwendung des angegebenen Puffers der Größe *sz* Bytes. Wenn *mode* gleich *ios:app* oder *ios::ate* ist, dann wird der Ein-/Ausgabezeiger auf das Nullzeichen des Strings positioniert. Im Kapitel 1 des *Programmierhandbuch* finden Sie eine Erläuterung zu den drei Zeichentypen.

Public-Elementfunktionen

str

```
char *str();
```

Liefert den Puffer zurück und verhindert weitere Pufferänderungen. Wenn der Puffer dynamisch war, dann müssen Sie ihn freigeben.

K A P I T E L 6

Klassen und Makros für persistente Streams

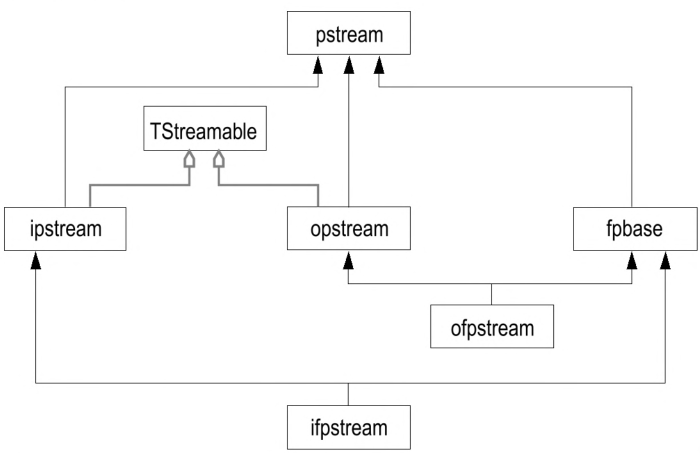
Details zur Verwendung der persistenten Stream-Bibliothek finden Sie im Kapitel 7 des *Programmierhandbuch.*

Borland unterstützt persistente Streams durch eine Klassenhierarchie und durch Makros, die Sie bei der Entwicklung streamfähiger Objekte unterstützen. In diesem Kapitel werden diese Klassen und Makros einzeln vorgestellt.

Alle öffentlichen Klassen für die Unterstützung von persistenten Objekten werden in alphabetischer Reihenfolge aufgeführt. Neben der Benutzung der Klasse werden ihre Elemente, Operatoren und Friend-Funktionen vorgestellt. Nach der Beschreibung der Klassen werden die Makros vorgestellt.

Die Klassenhierarchie für persistente Streams ist in der folgenden Abbildung dargestellt.

Abbildung 6.1: Klassenhierarchie für persistente Streams



Die Klasse *TStreamableBase* ist durch einen grauen Pfeil verbunden, da es sich um eine Friend-Klasse handelt.

Klasse fpbase objstrm.h

Stellt die grundlegenden Operationen zur Verfügung, die allen Objekten der Stream-Ein-/Ausgabe gemeinsam sind.

Public-Konstruktoren

```
fpbase();
fpbase(const char _FAR *name, int omode, int prot = filebuf::openprot);
fpbase(int f);
fpbase(int f, char _FAR *b, int len);
```

Erzeugt ein gepuffertes Objekt des Typs *fpbase*. Über die Argumente *len* und *b* können Sie die Größe und Initialisierung des Puffers bestimmen. Durch Angabe der Parameter für *name*, *omode* und *prot* für Dateinamen, Modus und Zugriffsrechte können Sie eine Datei öffnen und mit dem Stream verbinden. Sie können auch den Datei-Deskriptor *f* verwenden, um eine Datei mit dem Stream zu verbinden.

Public-Elementfunktionen

attach

```
void attach(int f);
```

Verbindet, wenn möglich, die Datei mit dem Deskriptor *f* mit dem Stream. Setzt *ios::state* entsprechend.

close

```
void close();
```

Schließt den Stream und die zugehörige Datei.

open

```
void open(const char _FAR *name, int mode, int prot = filebuf::openprot);
```

Öffnet die durch *name* angegebene Datei im Modus *mode* (*app*, *ate*, *in*, *out*, *binary*, *trunc*, *nocreate* oder *noreplace*) und mit den in *prot* angegebenen Zugriffsrechten. Die geöffnete Datei wird mit diesem Stream verbunden.

rdbuf

```
filebuf _FAR * rdbuf();
```

Liefert einen Zeiger auf den aktuellen Dateipuffer.

setbuf

```
void setbuf(char _FAR *buf, int len);
```

Setzt den Dateipuffer der Länge *len* auf die durch *buf* angegebene Adresse.

Klasse ifpstream objstrm.h

Bildet die Basisklasse für das Lesen von streamfähigen Objekten aus Datei-Streams.

Public-Konstruktoren

```
ifpstream();
ifpstream(const char _FAR *name, int mode=ios::in, int prot =
        filebuf::openprot);
ifpstream(int f);
ifpstream(int f, char _FAR *b, int len);
```

Erzeugt ein gepuffertes *ifpstream*-Objekt. Über die Argumente *len* und *b* können Sie die Größe und Initialisierung des Puffers bestimmen. Durch Angabe der Parameter für *name*, *omode* und *prot* für Dateinamen, Modus und Zugriffsrechte können Sie eine Datei öffnen und mit dem Stream verbinden. Sie können auch den Datei-Deskriptor *f* verwenden, um eine Datei mit dem Stream zu verbinden.

Public-Elementfunktionen

open

```
void open(const char _FAR *name, int mode = ios::in, int prot =
        filebuf::openprot);
```

Öffnet die durch *name* angegebene Datei im Modus *mode* (*app*, *ate*, *in*, *out*, *binary*, *trunc*, *nocreate* oder *noreplace*) und mit den in *prot* angegebenen Zugriffsrechten. Standardmäßig werden der Modus *in* (Eingabe) und die in *openprot* enthaltenen Zugriffsrechte verwendet. Die geöffnete Datei wird mit diesem Stream verbunden.

rdbuf

```
filebuf _FAR * rdbuf();
```

Liefert einen Zeiger auf den aktuellen Dateipuffer.

Klasse ipstream — objstrm.h

Bildet die Basisklasse für das Lesen von streamfähigen Objekten.

Public-Konstruktoren

```
ipstream(streambuf *buf);
```

Erzeugt ein gepuffertes *ipstream*-Objekt und setzt das Datenelement *bp* auf *buf*. Der Status (Datenelement *state*) wird auf 0 gesetzt.

Public-Elementfunktionen

find

```
TStreamableBase _FAR * find(P_id_type Id);
```

Liefert einen Zeiger auf das zu *Id* gehörende Objekt.

freadBytes

```
void freadBytes(void far *data, size_t sz);
```

Liest die durch *sz* angegebene Byte-Anzahl in den durch *data* bezeichneten **far**-Puffer.

freadString

```
char *freadString();
```

Liest einen String aus dem Stream.

Ermittelt die Länge des Strings und reserviert ein **far**-Zeichen-Array passender Länge. Liest den String in dieses Array ein und gibt einen Zeiger auf den String zurück. Die aufrufende Funktion sollte den reservierten Speicherblock wieder freigeben.

```
char far *freadString(char far *buf, unsigned maxLen);
```

Liest einen String aus dem Stream in den durch *buf* angegebenen **far**-Puffer ein. Ist die Stringlänge größer als *maxLen*-1, so wird nichts gelesen. Andernfalls wird der String in den Puffer eingelesen und ein abschließendes Nullzeichen angefügt.

getVersion

```
uint32 getVersion() const;
```

Liefert die Versionsnummer des Objekts.

readByte

```
uint8 readByte();
```

Liefert das Zeichen an der aktuellen Stream-Position.

readString

```
char _FAR * readString();
char _FAR * readString(char _FAR *buf, unsigned maxLen);
```

readString() reserviert einen ausreichend großen Puffer, um den String an der aktuellen Stream-Position aufzunehmen, und liest diesen String in den Puffer ein. Die aufrufende Funktion muß den Puffer wieder freigeben.

*readString(char _FAR *buf, unsigned maxLen)* liest den String an der aktuellen Stream-Position in den durch *puf* angegebenen Puffer ein. Es werden höchstens *maxLen* Bytes gelesen.

readWord

```
uint32 readWord();
```

Liefert das 32-Bit Wort an der aktuellen Stream-Position.

readWord16

```
uint16 readWord16();
```

Liefert das 16-Bit Wort an der aktuellen Stream-Position.

readWord

```
uint32 readWord();
```

Liefert das 32-Bit Wort an der aktuellen Stream-Position.

registerObject

```
void registerObject(TStreamableBase * adr);
```

Registriert das Objekt, auf das *adr* zeigt.

seekg

```
ipstream& seekg(streampos pos);
ipstream& seekg(streamoff off, ios::seekdir);
```

Die erste Form verschiebt die Stream-Position auf die durch *pos* angegebene absolute Adresse. Die zweite Form verschiebt die Stream-Position um einen durch *off* angegebenen Offset relativ zu der durch *ios::seek_dir* angegebenen Position. *ios::seek_dir* kann auf *beg* (Stream-Anfang), *cur* (aktuelle Stream-Position) oder *end* (Stream-Ende) gesetzt werden.

tellg

```
streampos tellg();
```

Liefert die (absolute) aktuelle Stream-Position.

Protected-Konstruktoren

```
ipstream();
```

Erzeugt ein gepuffertes ipstream-Objekt, ohne den Pufferzeiger *bp* zu initialisieren. Verwenden Sie *init*, um den Puffer und den Status zu setzen.

Protected-Elementfunktionen

readData

```
void _FAR * readData(const ObjectBuilder _FAR*, TStreamableBase _FAR *&mem);
```

Ruft die passende *read*-Funktion auf, um aus dem Stream das durch *mem* angegebene Objekt zu lesen. Wenn *mem* gleich 0 ist, wird zuerst die geeignete *build*-Funktion aufgerufen.

Siehe auch: *TStreamableClass* sowie die Elementfunktionen *read* und *build* der streamfähigen Klassen.

readPrefix

```
const ObjectBuilder _FAR * readPrefix();
```

Liefert ein *TStreamableClass*-Objekt zurück, das dem an der aktuellen Stream-Position gespeicherten Klassennamen entspricht.

readSuffix

```
void readSuffix();
```

Liest und überprüft das letzte Byte im Namensfeld eines Objektes.

Siehe auch: *ipstream::readPrefix*

readVersion

```
void readVersion();
```

Setzt die Versionsnummer des Eingabe-Streams.

Friend-Elemente

operator >>

```
friend ipstream& operator >> (ipstream& ps, signed char _FAR & ch);
friend ipstream& operator >> (ipstream& ps, unsigned char _FAR & ch);
friend ipstream& operator >> (ipstream& ps, signed short _FAR & sh);
friend ipstream& operator >> (ipstream& ps, unsigned short _FAR & sh);
friend ipstream& operator >> (ipstream& ps, signed int _FAR & i);
friend ipstream& operator >> (ipstream& ps, unsigned int _FAR & i);
friend ipstream& operator >> (ipstream& ps, signed long _FAR & l);
friend ipstream& operator >> (ipstream& ps, unsigned long _FAR & l);
friend ipstream& operator >> (ipstream& ps, float _FAR & f);
friend ipstream& operator >> (ipstream& ps, double _FAR & d);
friend ipstream& operator >> (ipstream& ps, long double _FAR & d);
friend ipstream& operator >> (ipstream& ps, TStreamableBase t);
friend ipstream& operator >> (ipstream& ps, void *t);
```

Liest aus dem *ipstream ps* in das gegebene Argument. Eine Referenz auf den Stream wird zurückgeliefert, so daß Sie mehrere **>>** Operationen auf die übliche Art verketten können. Der Datentyp des Arguments bestimmt, wie der Lesevorgang ausgeführt wird. Beispielsweise ist das Lesen eines **signed char** durch die Verwendung von *readByte* implementiert.

Klasse ofpstream — objstrm.h

Bildet die Basisklasse für das Schreiben von streamfähigen Objekten in Datei-Streams.

Public-Konstruktoren

```
ofpstream();
ofpstream(const char _FAR *name, int mode=ios::out,
          int prot = filebuf::openprot);
ofpstream(int f);
ofpstream(int f, char _FAR *b, int len);
```

Erzeugt ein gepuffertes *ofpstream*-Objekt. Über die Argumente *len* und *b* können Sie die Größe und Initialisierung des Puffers bestimmen. Durch Angabe der Parameter für *name*, *omode* und *prot* für Dateinamen, Modus und Zugriffsrechte können Sie eine Datei öffnen und mit dem Stream verbinden. Sie können auch den Datei-Deskriptor *f* verwenden, um eine Datei mit dem Stream zu verbinden.

Public-Elementfunktionen

open

```
void open(const char _FAR *name, int mode = ios::out,
         int prot = filebuf::openprot);
```

Öffnet die durch *name* angegebene Datei im Modus *mode* (*app*, *ate*, *in*, *out*, *binary*, *trunc*, *nocreate* oder *noreplace*) und mit den in *prot* angegebenen Zugriffsrechten. Standardmäßig werden der Modus *out* (Ausgabe) und die in *openprot* enthaltenen Zugriffsrechte verwendet. Die geöffnete Datei wird mit diesem Stream verbunden.

rdbuf

```
filebuf _FAR * rdbuf();
```

Liefert einen Zeiger auf den aktuellen Dateipuffer.

Klasse opstream — objstrm.h

opstream spezialisiert die Klasse *pstream* und ist selbst Basisklasse für das Schreiben streamfähiger Objekte.

Public-Konstruktoren

```
opsteam(streambuf _FAR *buf);
```

Erzeugt ein gepuffertes *opstream*-Objekt und setzt das Datenelement *bp* auf *buf*. Der Status (Datenelement *state*) wird auf 0 gesetzt.

Destruktor

```
~opstream();
```

Löscht das *opstream*-Objekt.

Siehe auch: *pstream::init*

Public-Elementfunktionen

findObject

```
P_id_type findObject(TStreamableBase _FAR *adr);
```

Liefert die Typ-ID des Objekts, auf das *adr* zeigt.

findVB

```
P_id_type findVB(TStreamableBase _FAR *adr);
```

Liefert einen Zeiger auf die virtuelle Basisklasse zurück.

flush

```
opstream& flush();
```

Leert den Stream-Puffer.

fwriteBytes

```
void fwriteBytes(const void far *data, size_t sz);
```

Schreibt die durch *sz* angegebene Byte-Anzahl in den durch *data* bezeichneten **far-**Puffer.

fwriteString

```
void *fwriteString(const char *str);
```

Schreibt den angegebenen String in den Stream.

registerObject

```
void registerObject(TStreamableBase _FAR * adr);
```

Registriert die Klasse des Objekts, auf das *adr* zeigt.

registerVB

```
void registerVB(TStreamableBase _FAR * adr);
```

Registriert eine virtuelle Basisklasse.

seekp

```
opstream& seekp(streampos pos);
opstream& seekp(streamoff off, ios::seekdir);
```

Die erste Form verschiebt die Stream-Position auf die durch *pos* angegebene absolute Adresse. Die zweite Form verschiebt die Stream-Position um einen durch *off* angegebenen Offset relativ zu der durch *ios::seek_dir* angegebenen Position. *ios::seek_dir* kann auf *beg* (Stream-Anfang), *cur* (aktuelle Stream-Position) oder *end* (Stream-Ende) gesetzt werden.

tellp

```
streampos tellp();
```

writeByte

```
void writeByte(uint8 ch);
```

Schreibt das Byte *ch* in den Stream.

writeBytes

```
void writeBytes(const void *data, size_t sz);
void writeBytes(const void far *data, size_t sz);
```

Schreibt *sz* Bytes aus dem Puffer *data* in den Stream.

writeObject

```
void writeObject(const TStreamableBase _BIDSFAR *t);
```

Schreibt das Objekt in den Stream, auf das *t* zeigt.

writeObjectPtr

```
void writeObjectPtr(const TStreamableBase *t);
```

Schreib den Objektzeiger *t* in den Ausgabe-Stream.

writeString

```
void writeString(const char _FAR *str);
```

Schreibt den durch *str* angegebenen String, einschließlich des führenden Längen-Bytes in den Stream.

writeWord

```
void writeWord(uint32 us);
```

Schreibt das 32-Bit Wort *us* in den Stream.

writeWord16

```
void writeWord16(uint16 us);
```

Schreibt das 16-Bit Wort *us* in den Stream.

writeWord32

```
void writeWord32(uint32 us);
```

Schreibt das 32-Bit Wort *us* in den Stream.

Protected-Konstruktoren

```
opstream();
```

Erzeugt ein gepuffertes *opstream*-Objekt, ohne den Pufferzeiger *bp* zu initialisieren. Verwenden Sie *init*, um den Puffer und den Status zu schreiben.

Protected-Elementfunktionen

writeData

```
void writeData(TStreamableBase *t);
```

Ruft die *write*-Elementfunktion der entsprechenden Klasse auf, um das Objekt in den Stream zu schreiben.

Siehe auch: *TStreamableBase* und die *write*-Funktionen aller streamfähigen Klassen.

writePrefix

```
void writePrefix(const TStreamableBase *t);
```

Schreibt das Klassennamen-Präfix in den Stream. Der Operator **<<** verwendet diese Funktion, um ein Präfix bzw. Suffix vor bzw. nach dem durch *writeData* geschriebenen Daten zu schreiben. Das Präfix bzw. Suffix wird verwendet, um eine typsichere Stream-Ein-/Ausgabe sicherzustellen.

Siehe auch: *opstream::readPrefix*

writeSuffix

```
void writeSuffix(const TStreamableBase *t);
```

Schreibt das Klassennamen-Suffix in den Stream. Der Operator **<<** verwendet diese Funktion, um ein Präfix bzw. Suffix vor bzw. nach den durch *writeData* geschriebenen Daten zu setzen. Präfix und Suffix werden verwendet, um eine typsichere Stream-Ein-/Ausgabe sicherzustellen.

Siehe auch: *opstream::readSuffix*

Friend-Elemente

operator <<

```
friend opstream& operator << (opstream& ps, signed char  ch);
friend opstream& operator << (opstream& ps, unsigned char ch);
friend opstream& operator << (opstream& ps, signed short sh);
friend opstream& operator << (opstream& ps, unsigned short sh);
friend opstream& operator << (opstream& ps, signed int i);
friend opstream& operator << (opstream& ps, unsigned int i);
friend opstream& operator << (opstream& ps, signed long l);
friend opstream& operator << (opstream& ps, unsigned long l);
friend opstream& operator << (opstream& ps, float f);
friend opstream& operator << (opstream& ps, double d);
friend opstream& operator << (opstream& ps, long double d);
friend opstream& operator << (opstream& ps, TStreamableBase& t);
```

Schreibt das angegebene Argument in das *ipstream*-Objekt. Der Datentyp des Arguments bestimmt die Form der durchgeführten Schreiboperation.

Klasse pstream — objstrm.h

pstream ist die Basisklasse für Verwaltung streamfähiger Objekte.

Typdefinitionen

PointerTypes

```
enum PointerTypes{ptNull, ptIndexed, ptObject};
```

Aufzählung von Objektzeiger-Typen.

Public-Konstruktoren

```
pstream(streambuf _FAR *buf);
```

Dieser Konstruktor erzeugt ein gepuffertes *pstream*-Objekt mit dem durch *buf* angegebenen Puffer. Das Datenelement *bp* wird auf *buf* und der Status (Datenelement *state*) wird auf 0 gesetzt.

Destruktor

```
virtual ~pstream();
```

Gibt das *pstream*-Objekt frei.

Public-Elementfunktionen

bad

```
int bad() const;
```

Liefert einen Wert ungleich null, wenn ein Fehler auftritt.

clear

```
void clear(int aState = 0);
```

Setzt den Status *state* des Streams auf den übergebenen Wert (standardmäßig auf 0).

eof

```
int eof() const;
```

Liefert einen Wert ungleich null, wenn das Ende des Streams erreicht ist.

fail

```
int fail() const;
```

Liefert einen Wert ungleich null, wenn bei einer vorhergehenden Stream-Operation ein Fehler aufgetreten ist.

good

```
int good() const;
```

Liefert einen Wert ungleich null, wenn keine Statusbits für den Stream gesetzt sind (also keine Fehler aufgetreten sind).

operator void *()

```
operator void *() const;
```

Überlädt den Typumwandlungsoperator Zeiger-auf-**void.** Liefert 0 zurück, wenn die Operation fehlgeschlagen ist, (d. h. *pstream::fail* lieferte einen Wert ungleich null zurück). Andernfalls liefert die Elementfunktion einen Wert ungleich null zurück.

Siehe auch: *pstream::fail*

operator !()

```
int operator !() const;
```

Überlädt den Negations-Operator (!) und liefert den von *pstream::fail* zurückgegebenen Wert zurück.

rdbuf

```
streambuf _FAR * rdbuf() const;
```

Liefert das Datenelement *bp* – den Zeiger auf den zum Stream gehörenden Puffer.

Siehe auch: *pstream::bp*

rdstate

```
int rdstate() const;
```

Liefert den aktuellen Wert des Datenelements *state*.

Protected-Datenelemente

bp

```
streambuf _FAR *bp;
```

Zeiger auf den Stream-Puffer.

state

```
int state;
```

Status-Flags. Verwenden Sie die Elementfunktion *rdstate*, um auf den aktuellen Status zuzugreifen.

Protected-Konstruktoren

```
pstream();
```

Dieser Konstruktor initialisiert den Pufferzeiger *bp* und den Status *state* nicht. Verwenden Sie *init* und *setstate*, um den Puffer und den Status zu setzen.

Protected-Elementfunktionen

init

```
void init(streambuf _FAR *sbp);
```

Initialisiert den Stream, indem *bp* auf *sbp* und *state* auf 0 gesetzt wird.

setstate

```
void setstate(int b);
```

Aktualisiert das Datenelement *state* mit `state |= (b & 0xFF)`.

Klasse TStreamableBase — objstrm.h

```
class _EXPCLASS TStreamableBase : public TCastable
```

Die von *TStreamableBase* abgeleiteten Klassen heißen streamfähige Klassen, da ihre Objekte in Streams geschrieben bzw. aus Streams gelesen werden können. Möchten Sie eigene streamfähige Klassen entwickeln, so sorgen Sie dafür, daß sie in irgendeiner Form von *TStreamableBase* abgeleitet sind. Am einfachsten erreicht man dies natürlich, indem man eine bestehende streamfähige Klasse als Basis der eigenen Klasse verwendet. Scheuen Sie sich nicht vor der Verwendung von Mehrfachvererbungen, wenn ihre Klasse zusätzlich in eine bestehende Klassenhierarchie eingegliedert werden muß.

Typdefinitionen

Type_id

```
typedef const char *Type_id
```

Beschreibt Typ-IDs.

Destruktor

```
virtual ~TStreamableBase() {};
```

Gibt ein *TStreamableBase*-Objekt wieder frei.

Public-Elementfunktionen

CastableID

```
virtual Type_id CastableID() const = 0;
```

Unterstützt sichere Typumwandlungen und liefert einen String mit dem Typnamen zurück.

FindBase

```
virtual void *FindBase(Type_id id) const;
```

Liefert einen Zeiger auf die Basisklasse.

MostDerived

```
virtual void *MostDerived() const = 0;
```

Klasse TStreamableClass — streambl.h

TStreamableClass wird von der als privat deklarierten Datenbank-Klasse und von *pstream* für die Registrierung streamfähiger Klassen verwendet.

Public-Konstruktoren

```
TStreamableClass(const char *n, BUILDER b, int d=NoDelta, ModuleId
                 mid=GetModuleId());
```

Erzeugt ein *TStreamableClass*-Objekt mit dem Namen *n* und der *build*-Funktion *b* und registriert den Typ. Jede streamfähige Klasse (beispielsweise *TClassname*) hat eine *build*-Elementfunktion vom Typ BUILDER. Um eine typsichere Objekt-Ein-/Ausgabe auf Streams zu gewährleisten, benötigt die Stream-Verwaltung Zugriff auf die Namen sowie die Typinformationen jeder Klasse. Um sicherzustellen, daß die entsprechenden Funktionen zu jeder Anwendung die Streams benutzt, hinzugelinkt werden, müssen Sie etwa folgendes in Ihr Programm aufnehmen:

```
TStreamableClass RegClassName;
```

wobei *TClassName* der Name der Klasse ist, deren Objekte streamfähig sein sollen. (Beachten Sie, daß *RegClassName* ein einziger Bezeichner ist.) Dadurch wird nicht nur *TClassName* registriert (der Stream-Verwaltung wird die zu verwendende *build*-Funktion mitgeteilt), sondern es werden automatisch auch alle abgeleiteten Klassen registriert. Sie können eine Klasse problemlos mehrfach registrieren.

Rufen Sie diese Funktion auf, damit für das Einlesen von Objekten der Klasse ein genügend großer Speicherplatz bereitgestellt werden kann. Da die *build*-Prozedur einen speziellen Konstruktor für die Klasse aufruft, werden alle Zeiger der virtuellen Tabelle korrekt initialisiert.

Die Distanz in Byte zwischen der Basis des streamfähigen Objekts und dem Anfang der *TStreamable*-Komponente des Objekts ist *d*. Sie können *d* errechnen, indem Sie das Makro __DELTA wie im folgenden Beispiel verwenden:

```
TStreamableClass RegTClassName = TStreamableClass("TClassName",
TClassName::build, __DELTA(TClassName));
```

Siehe auch: *TStreamableBase*, *ipstream*, *opstream*

Friend-Klassen

Die Klassen *opstream* und *ipstream* sind Friends von *TStreamableClass*.

Klasse TStreamer — objstrm.h

```
class _BIDSCLASS _RTTI TStreamer
```

Basisklasse aller streamfähigen Objekte.

Public-Elementfunktionen

GetObject

```
TStreamableBase *GetObject() const;
```

Liefert die Adresse der *TStreamableBase*-Komponente des streamfähigen Objekts zurück.

Protected-Konstruktoren

```
TStreamer(TStreamableBase *obj);
```

Erzeugt ein *TStreamer*-Objekt und initialisiert den Zeiger auf das streamfähige Objekt.

Protected-Elementfunktionen

Read

```
virtual void *Read(ipstream&, uint32) const = 0;
```

Diese abstrakte Elementfunktion muß in jeder streamfähigen Klasse überschrieben werden. Die Funktion muß die benötigten Datenelemente der streamfähigen Klasse aus dem durch *ipstream* angegebenen Stream lesen.

StreamableName

```
virtual const char *StreamableName() const = 0;
```

Diese abstrakte Elementfunktion muß in jeder streamfähigen Klasse überschrieben werden. *StreamableName* liefert den Namen der streamfähigen Klasse zurück, der von der Stream-Verwaltung für die Registrierung der streamfähigen Klasse verwendet wird. Der zurückgegebene Name muß ein nullterminierter String sein.

Write

```
virtual void Write(opstream&) const = 0;
```

Diese abstrakte Elementfunktion muß in jeder streamfähigen Klasse überschrieben werden. Die Funktion muß die benötigten Datenelemente der streamfähigen Klasse in den durch *opstream* angegebenen Stream schreiben. *Write* wird normalerweise so implementiert, daß zunächst die *Write*-Funktion der Basisklasse aufgerufen wird (sofern sie existiert) und danach alle zusätzlichen Datenelemente der abgeleiteten Klasse geschrieben werden.

Makro __DELTA — streambl.h

```
#define __DELTA(d) (FP_OFF((TStreamable *)(d *)1)-1)
```

Makro zur Berechnung der Distanz in Byte vom Beginn des Speicherbereichs eines streamfähigen Objekts bis zur *TStreamableBase*-Komponente dieses Objekts. Argument ist der Name der streamfähigen Klasse.

Stream-Makros

Die im folgenden vorgestellten Makros vereinfachen die Deklaration und Definition von streamfähigen Klassen.

Makro DECLARE_STREAMABLE objstrm.h

```
DECLARE_STREAMABLE(exp, cls, ver)
```

Das Makro DECLARE_STREAMABLE wird innerhalb einer Klassendefinition benutzt, um die für Streams benötigten Elemente hinzuzufügen. Da im Makro Zugriffs-Spezifizierer enthalten sind, sollte das Makro entweder am Ende der Klassendefinition benutzt oder aber von einem Zugriffs-Spezifizierer gefolgt werden. Der erste Parameter sollte ein Makro sein, das zu *__import* oder *__export* erweitert wird, je nachdem, ob die Klasse aus einer DLL importiert oder exportiert wird. Der zweite Parameter von DECLARE_STREAMABLE ist der Name der streamfähigen Klasse. Der dritte ist die Objekt-Versionsnummer.

DECLARE_STREAMABLE ist folgendermaßen definiert:

```
#define DECLARE_STREAMABLE(exp, cls, ver)          \
    DECLARE_CASTABLE                               \
    DECLARE_STREAMER(exp, cls, ver);               \
    DECLARE_STREAMABLE_OPS(cls);                   \
    DECLARE_STREAMBALE_CTOR(cls)
```

Makro DECLARE_STREAMABLE_FROM_BASE objstrm.h

```
DECLARE_STREAMABLE_FROM_BASE(exp, cls, ver)
```

DECLARE_STREAMABLE_FROM_BASE erfüllt dieselbe Funktion wie DECLARE_STREAMABLE, wird jedoch immer dann eingesetzt, wenn die definierte Klasse die Elementfunktionen *Read* und *Write* ihrer Basisklasse ohne Änderung benutzen kann. Dies ist normalerweise dann der Fall, wenn in der abgeleiteten Klasse virtuelle Funktionen der Basisklasse überschrieben oder weitere Konstruktoren definiert, jedoch keine zusätzlichen Datenelemente hinzugefügt werden. Wenn Sie in einem solchen Fall DECLARE_STREAMABLE aufrufen, denn müßten Sie *Read* und *Write* Funktionen schreiben, die lediglich die Funktionen *Read* und *Write* der Basisklasse aufrufen. Das Makro DECLARE_STREAMABLE_FROM_BASE macht dies überflüssig.

DECLARE_STREAMABLE_FROM_BASE ist folgendermaßen definiert:

```
#define DECLARE_STREAMABLE_FROM_BASE(exp, cls, base)  \
    DECLARE_CASTABLE                                  \
    DECLARE_STREAMER_FROM_BASE(exp, cls, base);       \
    DECLARE_STREAMABLE_OPS(cls);                      \
    DECLARE_STREAMBALE_CTOR(cls)
```

Makro DECLARE_ABSTRACT_STREAMABLE objstrm.h

```
DECLARE_ABSTRACT_STREAMABLE(exp, cls, ver)
```

Dieses Makro wird in abstrakten Klassen eingesetzt, in denen DECLARE_STREAMABLE nicht benutzt werden kann. Da abstrakte Klassen nicht instantiiert werden können, würde der Programmcode der Funktion *Build*, der versucht das Objekt zu instantiieren, einen Compilerfehler auslösen. Das Makro wird zu DECLARE_CASTABLE, DECLARE_ABSTRACT_STREAMER, DECLARE_STREAMABLE_OPS und DECLARE_STREAMABLE_CTOR expandiert.

Makro DECLARE_STREAMER objstrm.h

```
DECLARE_STREAMER(exp, cls, ver)
```

Diese Makro definiert innerhalb Ihrer streamfähigen Klasse eine weitere Klasse und enthält den Kern des Stream-Programmcodes. DECLARE_STREAMER deklariert die Funktionen *Read* und *Write*, deren Definitionen Sie bereitstellen müssen, sowie die Funktion *Build*, die den Konstruktor von *TStreamableClass* aufruft. Eine Erklärung der Parameter finden Sie unter DECLARE_STREAMABLE.

Makro DECLARE_STREAMER_FROM_BASE objstrm.h

```
DECLARE_STREAMER_FROM_BASE(exp, cls, base)
```

Dieses Makro wird von DECLARE_STREAMABLE_FROM_BASE benutzt. Es deklariert eine geschachtelte *Streamer*-Klasse ohne die Funktionen *Read* und *Write*. Eine Erklärung der Parameter finden Sie unter DECLARE_STREAMABLE.

Makro DECLARE_ABSTRACT_STREAMER objstrm.h

```
DECLARE_ABSTRACT_STREAMER(exp, cls, ver)
```

Dieses Makro wird von DECLARE_ABSTRACT_STREAMABLE benutzt. Es deklariert eine geschachtelte *Streamer*-Klasse ohne die Funktion *Build*. Eine Erklärung der Parameter finden Sie unter DECLARE_STREAMABLE.

Makro DECLARE_CASTABLE objstrm.h

```
DECLARE_CASTABLE
```

Dieses Makro enthält die Deklarationen für die grundlegenden Verfahren zur sicheren Typumwandlung. Dies ist hilfreich bei Compilern, die Typinformationen zur Laufzeit nicht unterstützen.

Makro DECLARE_STREAMABLE_OPS objstrm.h

```
DECLARE_STREAMABLE_OPS(cls)
```

Deklariert die Operatoren **<<** und **>>**. In Template-Klassen muß DECLARE_STREAMABLE_OPS `class...` als Argument übernehmen – die übrigen DECLARE-Makros übernehmen nur den Klassennamen.

Makro DECLARE_STREAMABLE_CTOR objstrm.h

```
DECLARE_STREAMABLE_CTOR(cls)
```

Deklariert den von der Funktion *Streamer::Build* aufgerufenen Konstruktor.

Makros IMPLEMENT_STREAMABLE objstrm.h

```
IMPLEMENT_STREAMABLE(cls)
IMPLEMENT_STREAMABLE1(cls, base1)
IMPLEMENT_STREAMABLE2(cls, base1, base2)
IMPLEMENT_STREAMABLE3(cls, base1, base2, base3)
IMPLEMENT_STREAMABLE4(cls, base1, base2, base3, base4)
IMPLEMENT_STREAMABLE5(cls, base1, base2, base3, base4, base5)
```

Die IMPLEMENT_STREAMABLE-Makros generieren mit Hilfe von IMPLEMENT_STREAMABLE_CLASS das Registrierungsobjekt für die Klasse. Mit IMPLEMENT_ABSTRACT_STREAMABLE werden verschiedene Elementfunktionen erzeugt, die für eine streamfähige Klasse benötigt werden.

IMPLEMENT_STREAMABLE wird benutzt, wenn die Klasse außer *TStreamableBase* keine weiteren Basisklassen hat. Der einzige Makroparameter ist der Klassenname. Die numerierten Makroversionen (IMPLEMENT_STREAMABLE1, IMPLEMENT_STREAMABLE2 usw.) sind für Klassen, die neben *TStreamableBase* weitere Basisklassen besitzen. Jede Basisklasse – inklusive aller virtuellen Basisklassen – muß im Aufruf des entsprechenden IMPLEMENT_STREAMABLE-Makros aufgeführt werden.

Die einzelnen Komponenten, aus denen sich die obigen Makros zusammensetzen, können in besonderen Situationen auch separat benutzt werden, um beispielsweise einen speziellen Konstruktor zu definieren.

Makro IMPLEMENT_STREAMABLE_CLASS — objstrm.h

```
IMPLEMENT_STREAMABLE_CLASS(cls)
```

Erzeugt eine Instanz von *TStreamableClass*.

Makros IMPLEMENT_STREAMABLE_CTOR — objstrm.h

```
IMPLEMENT_STREAMABLE_CTOR(cls)
IMPLEMENT_STREAMABLE_CTOR1(cls, base1)
IMPLEMENT_STREAMABLE_CTOR2(cls, base1, base2)
IMPLEMENT_STREAMABLE_CTOR3(cls, base1, base2, base3)
IMPLEMENT_STREAMABLE_CTOR4(cls, base1, base2, base3, base4)
IMPLEMENT_STREAMABLE_CTOR5(cls, base1, base2, base3, base4, base5)
```

Definiert den von der Funktion *Build* aufgerufenen Konstruktor. Alle Basisklassen müssen im entsprechenden Makroaufruf aufgeführt werden.

Makro IMPLEMENT_STREAMABLE_POINTER — objstrm.h

```
IMPLEMENT_STREAMABLE_POINTER(cls)
```

Implementiert den Leseoperator >>.

Makro IMPLEMENT_CASTABLE_ID — objstrm.h

```
IMPLEMENT_CASTABLE_ID(cls)
```

Legt die ID zur sicheren Typumwandlung fest.

Makros IMPLEMENT_CASTABLE objstrm.h

```
IMPLEMENT_CASTABLE(cls)
IMPLEMENT_CASTABLE1(cls)
IMPLEMENT_CASTABLE2(cls)
IMPLEMENT_CASTABLE3(cls)
IMPLEMENT_CASTABLE4(cls)
IMPLEMENT_CASTABLE5(cls)
```

Diese Makros erzeugen den Programmcode, der die Verfahren zur sicheren Typumwandlung unterstützt.

Makro IMPLEMENT_STREAMER objstrm.h

```
IMPLEMENT_STREAMER(cls)
```

Definiert den *Streamer*-Konstruktor.

Makros IMPLEMENT_ABSTRACT_STREAMABLE objstrm.h

```
IMPLEMENT_ABSTRACT_STREAMABLE(cls)
IMPLEMENT_ABSTRACT_STREAMABLE1(cls)
IMPLEMENT_ABSTRACT_STREAMABLE2(cls)
IMPLEMENT_ABSTRACT_STREAMABLE3(cls)
IMPLEMENT_ABSTRACT_STREAMABLE4(cls)
IMPLEMENT_ABSTRACT_STREAMABLE5(cls)
```

Diese Makros werden zu IMPLEMENT_STREAMER (definiert den Konstruktor für *Streamer*), IMPLEMENT_STREAMABLE_CTOR (definiert den Konstruktor für *TStreamableClass*) und IMPLEMENT_STREAMABLE_POINTER (definiert den Operator >>) expandiert.

Makro IMPLEMENT_STREAMABLE_FROM_BASE objstrm.h

```
IMPLEMENT_STREAMABLE_FROM_BASE(cls, base1)
```

Dieses Makro wird zu IMPLEMENT_STREAMABLE_CLASS (erzeugt eine Instanz von *TStreamableClass*), IMPLEMENT_STREAMABLE_CTOR1 (definiert einen Konstruktor für eine Basisklasse, der von *Build* aufgerufen wird) und IMPLEMENT_STREAMABLE_POINTER (definiert den Operator >>) expandiert.

K A P I T E L 7

Die C++ Containerklassen

Im Kapitel 7 des *Programmierhandbuch* finden Sie weitere Informationen zur Benutzung von Containerklassen.

Dieses Kapitel beinhaltet eine Übersicht über die Borland C++ Containerklassen. Jede Containerklasse gehört zu einer der im folgenden zusammen mit der entsprechenden Header-Datei aufgeführten Gruppen:

- Array (arrays.h)
- Association (assoc.h)
- Bag (bags.h)
- Association (assoc.h)
- Binärbaum (binimp.h)
- Dequeue (deques.h)
- Dictionary (dict.h)
- Doppelt verkettete Liste (dlistimp.h)
- Hash-Tabelle (hashimp.h)
- Liste (listimp.h)
- Queue (queues.h)
- Set (sets.h)
- Stack (sets.h)
- Vektor (vectimp.h)

Template TMArrayAsVector arrays.h

TMArrayAsVector realisiert ein Array mit eigener Speicherverwaltung von Objekten des Typs *T*, wobei als zugrundeliegende Datenstruktur ein Vektor verwendet wird. Für den Typ *T* wird dazu der Operator == benötigt.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef int ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

```
TMArrayAsVector(int upper, int lower = 0, int delta = 0);
```

Erzeugt ein Array mit der oberen bzw. unteren Grenze *upper* bzw. *lower*, und dem Größenzuwachswert *delta*.

Public-Funktionstypen

Add

```
int Add(const T& t)
```

Fügt ein *T*-Objekt an der nächsten freien Indexposition ein. Bei Überschreiten der oberen Indexgrenze entsteht ein Überlauf, der zu einer Vergrößerung des Arrays um *delta* Einträge führt, sofern der Größenzuwachswert *delta* größer als 0 ist. Wenn *delta* auf 0 gesetzt ist, wird in diesem Fall ein Fehler ausgelöst.

AddAt

```
int AddAt(const T& t, int loc)
```

Nimmt ein *T*-Objekt an der durch *loc* angegebenen Indexposition in das Array auf. Ist diese Position bereits belegt, so wird das bisherige Element um eine Position noch oben verschoben. Befindet sich die Position hinter der oberen Grenze, so wird das Array erweitert, sofern *delta* (siehe Konstruktor) größer als 0 ist. Bei *delta* == 0 führt der Versuch, mit *AddAt* hinter der oberen Grenze ein Element einzufügen, zu einem Fehler.

ArraySize

```
unsigned ArraySize() const
```

Liefert die Array-Größe als Anzahl der reservierten Zellen.

Destroy

```
int Destroy(int i)
```

Entfernt das Objekt mit dem angegebenen Index. Das Objekt wird freigegeben.

Detach

`int Detach(int loc, TShouldDelete::DeleteType dt = TShouldDelete::NoDelete)`
`int Detach(const T& t, TShouldDelete::DeleteType dt = TShouldDelete::NoDelete)`

Die erste Version entfernt das Objekt an der Indexposition *loc* aus dem Array. Die zweite Version entfernt das erste Objekt aus dem Array, das dem angegebenen Objekt gleicht. Der Wert von *dt* sowie der aktuelle Besitzerstatus bestimmen, ob das Objekt selbst gelöscht wird. *DeleteType* ist in der Basisklasse *TShouldDelete* als enum {NoDelete, DefDelete, Delete} aus definiert. Der Standardwert von *dt* ist *NoDelete* und bewirkt, daß das Objekt unabhängig vom Besitzer nicht gelöscht wird. Wird *dt* auf *Delete* gesetzt, so wird das Objekt unabhängig vom Besitzer in jedem Fall gelöscht. Enthält *dt* den Wert *DefDelete*, so wird das Objekt nur dann gelöscht, wenn das Array Besitzer seiner Elemente ist.

Siehe auch: *TShouldDelete:ownsElements*

FirstThat

`T *FirstThat(CondFunc f, void *), void *args) const`

Liefert einen Zeiger auf das erste Array-Objekt, das einer gegebenen Bedingung genügt. Sie müssen den Zeiger *f* auf eine Testfunktion angeben, die einen Wert ungleich 0 liefert, wenn eine bestimmte Bedingung eingetreten ist. Über den Parameter *args* können beliebige Argumente übergeben werden. Wird die Bedingung von keinem Objekt im Array erfüllt, so liefert die Funktion den Wert 0 zurück.

Siehe auch: *LastThat*

Flush

`void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)`

Entfernt alle Elemente aus dem Array, ohne jedoch das Array zu löschen. Der Wert von *dt* bestimmt, ob die Elemente selbst gelöscht werden. Standardmäßig wird das »Schicksal« der Elemente vom Besitzerstatus bestimmt (siehe Erläuterungen zur Elementfunktion *Detach*). Sie können *dt* auch auf *Delete* oder *Nodelete* setzen.

Siehe auch: *Detach*

ForEach

`void ForEach(IterFunc f, void *args)`

ForEach erzeugt einen internen Iterator, mit dem die angegebene Funktion auf jedes Array-Element angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

`unsigned GetItemsInContainer() const`

Liefert die Anzahl der Elemente im Array zurück (im Gegensatz zu *ArraySize*, das die Array-Größe zurückliefert).

HasMember

`int HasMember(const T& t) const`

Liefert, wenn das gegebene Objekt im Array gefunden werden konnte, den Wert 1, andernfalls den Wert 0 zurück.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls das Array kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 1 zurück, falls das Array voll ist. Andernfalls wird 0 zurückgeliefert. Das Array ist voll, wenn *delta* ungleich null und Anzahl der Elemente im Array gleich dem von *ArraySize* zurückgelieferten Wert ist.

LastThat `T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Objekt des Array zurück, das eine gegebene Bedingung erfüllt. Der Zeiger auf die Testfunktion *f*, die für eine bestimmte Bedingung einen Wert ungleich 0 zurückliefert, wird von Ihnen bereitgestellt. Über den Parameter *args* können Sie beliebige Argumente übergeben. Erfüllt kein Array-Objekt die Bedingung, so wird der Wert 0 zurückgeliefert. Beachten Sie, daß *LastThat* einen eigenen internen Iterator erzeugt, so daß Sie es wie eine »Suchfunktion« benutzen können.

Siehe auch: *FirstThat*, *ForEach*

LowerBound `int LowerBound() const`

Liefert die untere Array-Grenze *lowerbound* zurück.

Upperbound `int UpperBound() const`

Liefert die obere Array-Grenze *upperbound* zurück.

Protected-Elementfunktionen

BoundBase `int BoundBase(unsigned loc) const`

Vektoren haben im Gegensatz zu Arrays 0 als untere Grenze. Mit *BoundBase* werden Vektoren an Arrays angeglichen. Siehe *ZeroBase*.

Find `int Find(const T& t) const`

Sucht das angegebene Objekt und liefert seine Indexposition zurück, wenn es gefunden wurde. Andernfalls wird INT_MAX zurückgeliefert.

Grow `void Grow(int loc)`

Erweitert das Array so, daß *loc* eine gültige Indexposition ist.

InsertEntry `void InsertEntry(int loc)`

Erzeugt ein Objekt und fügt es an der Position *loc* ein, indem die darüberliegenden Einträge um eine Position nach oben geschoben werden.

ItemAt `T ItemAt(int i) const`

Liefert eine Kopie des an der Position *i* gespeicherten Objekts zurück.

Reallocate `int Reallocate(unsigned sz, unsigned offset = 0)`

Ist das Datenelement *delta* (siehe Konstruktor) gleich 0, dann liefert *Reallocate* den Wert 0 zurück. Ansonsten versucht die Funktion, ein neues Array der Größe *sz* anzulegen (aufgerundet auf das nächste Vielfache von *delta*). Das bestehende Array wird in das erweiterte Array kopiert und danach gelöscht. In einem Pointer-Array werden Einträge für unbenutzte Elemente auf NULL gesetzt. In einem Objekt-Array wird für jedes unbenutzte Element der Standardkonstruktor aufgerufen. Der Parameter *offset* bezeichnet die Position im neuen Vektor, die das erste Element des alten Vektors aufnehmen soll. Dies wird benötigt, wenn das Array nach unten erweitert werden soll.

RemoveEntry `void RemoveEntry(int loc)`

Entfernt das Element an Position *loc* und verkleinert das Array um ein Element. Die Elemente an den Indexpositionen ab (*loc* + 1) werden um eine Position nach unten verschoben. Das alte Element an Position *loc* wird gelöscht.

SetData `void SetData(int loc, const T& t)`

Das gegebene Element *t* ersetzt das aktuelle Element an Position *loc*.

SqueezeEntry `void SqueezeEntry(unsigned loc)`

Entfernt das Element an Position *loc* und verkleinert das Array um ein Element. Die Elemente an den Indexpositionen ab (*loc* + 1) werden um eine Position nach unten verschoben. Das alte Element an Position *loc* wird gelöscht.

ZeroBase `unsigned ZeroBase(int loc) const`

Liefert die Position relativ zu *lowerbound* zurück (das entspricht dem Wert *loc - lowerbound*).

Operatoren

operator []

```
T& operator [] (int loc)
T& operator [] (int loc) const
```

Liefert eine Referenz auf das Element an Position *loc* zurück. Die erste Form (ohne **const**) erweitert das Array falls nötigt, damit *loc* eine gültige Indexposition ist. Die zweite Version (mit **const**) löst beim versuchten Zugriff auf Positionen außerhalb der Indexgrenzen eine Exception aus.

Template TMArrayAsVectorIterator arrays.h

Implementiert ein Iterator-Objekt, mit dem *TMArrayAsVector*-Objekte durchlaufen werden können.

Konstruktoren

`TMArrayAsVectorIterator( const TMArrayAsVector<T,Alloc> & a) :`

Erzeugt ein Iterator-Objekt, mit dem *TMArrayAsVector*-Objekte durchlaufen werden.

Public-Elementfunktionen

Current

`const T& Current();`

Liefert das aktuelle Objekt zurück.

Restart

```
void Restart();
void Restart(unsigned start, unsigned stop);
```

Beginnt das Durchlaufen von vorne oder innerhalb des angegebenen Bereichs.

Operatoren

operator ++

`const T& operator ++(int);`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

`const T& operator ++();`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

operator int

`operator int() const`

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

Template TArrayAsVector arrays.h

TArrayAsVector implementiert auf Basis eines Vektors ein Array von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Klassenelemente finden Sie unter *TMArrayAsVector*.

Konstruktoren

```
TArrayAsVector(int upper, int lower =0, int delta = 0)
```

Erzeugt ein Array mit der oberen Grenze *upper*, der unteren Grenze *lower* und einem Größenzuwachs von *delta*.

Template TArrayAsVectorIterator — arrays.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TArrayAsVector*-Objekten. Die Elementfunktionen und Operatoren finden Sie unter *TMArrayAsVectorIterator*.

Konstruktoren

```
TArrayAsVectorIterator(const TArrayAsVector<T> & a)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TArrayAsVector*-Objekten.

Template TMIArrayAsVector — arrays.h

Implementiert auf Basis eines Vektors ein indirektes Array mit eigener Speicherverwaltung von Objekten des Typs *T*.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef int ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

```
TMIArrayAsVector(int upper, int lower = 0, int delta = 0)
```

Erzeugt ein indirektes Array mit der oberen Grenze *upper*, der unteren Grenze *lower* und einem Größenzuwachs von *delta*.

Public-Funktionstypen

Add

`int Add(T *t)`

Fügt einen Zeiger auf ein *T*-Objekt an die nächste freie Indexposition ein. Bei Hinzufügen eines Elements jenseits der oberen Grenze entsteht ein Überlauf, der entweder zu einer Vergrößerung des Array (falls *delta* ungleich 0) um geeignete Vielfache von *delta* Bytes oder zu einem Fehler führt (falls *delta* gleich 0). Ist *delta* gleich 0, dann endet *Add* mit einem Fehler. Wenn das Objekt nicht hinzugefügt werden kann, liefert *Add* den Wert 0 zurück.

AddAt

`int AddAt(T *t, int loc)`

Fügt einen Zeiger auf ein *T*-Objekt an der angegebenen Indexposition ein. Ist die Position belegt, so wird das bisherige Objekt gelöscht. Befindet sich *loc* jenseits der oberen Grenze, wird das Array vergrößert sofern *delta* ungleich 0 ist (siehe Konstruktor). Ist *delta* gleich 0, so tritt bei einer *AddAt*-Operation jenseits der oberen Grenze ein Fehler auf.

ArraySize

`unsigned ArraySize() const`

Liefert die aktuelle Anzahl reservierter Zellen.

Destroy

`int Destroy(int i)`

Entfernt das Objekt mit dem angegebenen Index. Das Objekt wird freigegeben.

`int Destroy(T *t)`

Entfernt das durch *t* angegebene Objekt und gibt es frei.

Detach

```
int Detach(int loc, DeleteType dt = NoDelete)
int Detach(T*t, DeleteType dt = NoDelete)
```

Die erste Version entfernt das Objekt an der Indexposition *loc* aus dem Array. Die zweite Version entfernt den angegebenen Zeiger aus dem Array. Der Wert von *dt* sowie der aktuelle Besitzerstatus bestimmen, ob das Objekt selbst gelöscht wird. *DeleteType* ist in der Basisklasse *TShouldDelete* als `enum {NoDelete, DefDelete, Delete}` definiert. Der Standardwert von *dt* ist *NoDelete* und bewirkt, daß das Objekt unabhängig vom Besitzer nicht gelöscht wird. Wird *dt* auf *Delete* gesetzt, so wird das Objekt unabhängig vom Besitzer in jedem Fall gelöscht. Enthält *dt* den Wert *DefDelete*, so wird das Objekt nur dann gelöscht, wenn das Array Besitzer seiner Elemente ist.

Siehe auch: *TShouldDelete::ownsElements*

FirstThat

`T *FirstThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das erste Array-Objekt, das einer gegebenen Bedingung genügt. Sie müssen den Zeiger *f* auf eine Testfunktion angeben, die einen Wert ungleich 0 liefert, wenn eine bestimmte Bedingung einge-

treten ist. Über den Parameter *args* können beliebige Argumente übergeben werden. Wird die Bedingung von keinem Objekt im Array erfüllt, so liefert die Funktion den Wert 0. Beachten Sie, daß *FirstThat* einen eigenen internen Iterator verwendet, und Sie die Funktion daher wie eine »Suchfunktion« auffassen können.

Siehe auch: *LastThat*

Find

`int Find(const T t) const`

Sucht das angegebene Objekt und liefert seine Indexposition zurück, wenn es gefunden wurde. Andernfalls wird INT_MAX zurückgeliefert.

Flush

`void Flush(DeleteType dt = DefDelete)`

Entfernt alle Elemente aus dem Array, ohne jedoch das Array zu löschen. Der Wert von *dt* bestimmt, ob die Elemente selbst gelöscht werden. Standardmäßig wird das »Schicksal« der Elemente vom Besitzerstatus des Arrays bestimmt (siehe Erläuterungen zur Elementfunktion *Detach*). Sie können *dt* auch auf *Delete* oder *Nodelete* setzen.

Siehe auch: *Detach*

ForEach

`void ForEach(IterFunc f, void *args)`

ForEach erzeugt einen internen Iterator, mit dem die angegebene Funktion auf jedes Array-Element angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

`unsigned GetItemsInContainer() const`

Liefert die Anzahl der Elemente im Array zurück.

HasMember

`int HasMember(const T& t) const`

Liefert, wenn das gegebene Objekt im Array gefunden werden konnte, den Wert 1, andernfalls den Wert 0 zurück.

IsEmpty

`int IsEmpty() const`

Liefert den Wert 1 zurück, falls das Array kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull

`int IsFull() const`

Liefert den Wert 1 zurück, falls das Array voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat

`T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Objekt des Array zurück, das eine gegebene Bedingung erfüllt. Der Zeiger auf die Testfunktion *f*, die für eine bestimmte Bedingung einen Wert ungleich 0 zurückliefert, wird von Ihnen bereitgestellt. Über den Parameter *args* können Sie beliebige Argumente übergeben. Erfüllt kein Array-Objekt die Bedingung, so wird der Wert 0 zurückgeliefert. Beachten Sie, daß *LastThat* einen eigenen internen Iterator erzeugt, so daß Sie es wie eine »Suchfunktion« benutzen können.

Siehe auch: *FirstThat*, *ForEach*

LowerBound `int LowerBound() const`

Liefert die untere Array-Grenze *lowerbound* zurück.

Upperbound `int UpperBound() const`

Liefert die obere Array-Grenze *upperbound* zurück.

Protected-Elementfunktionen

BoundBase `int BoundBase(unsigned loc) const`

Vektoren haben im Gegensatz zu Arrays 0 als untere Grenze. Mit *BoundBase* werden Vektoren an Arrays angeglichen. Siehe *ZeroBase*.

Grow `void Grow(int loc)`

Erweitert das Array so, daß *loc* eine gültige Indexposition ist.

InsertEntry `void InsertEntry(int loc)`

Erzeugt ein Objekt und fügt es an der Position *loc* ein.

ItemAt `T ItemAt(int i) const`

Liefert eine Kopie des an der Position *i* gespeicherten Objekts zurück.

Reallocate `int Reallocate(unsigned sz, unsigned offset = 0)`

Ist das Datenelement *delta* (siehe Konstruktor) gleich 0, dann liefert *Reallocate* den Wert 0 zurück. Ansonsten versucht die Funktion, ein neues Array der Größe *sz* anzulegen (aufgerundet auf das nächste Vielfache von *delta*). Das bestehende Array wird in das erweiterte Array kopiert und danach gelöscht. In einem Pointer-Array werden Einträge für unbenutzte Elemente auf NULL gesetzt. In einem Objekt-Array wird für jedes unbenutzte Element der Standardkonstruktor aufgerufen. Der Parameter *offset* bezeichnet die Position im neuen Vektor, die das erste Element des alten Vektors aufnehmen soll. Dies wird benötigt, wenn das Array nach unten erweitert werden soll.

RemoveEntry `void RemoveEntry(int loc)`

Entfernt das Element an Position *loc* und verkleinert das Array um ein Element. Die Elemente an den Indexpositionen ab (*loc* + 1) werden um eine Position nach unten verschoben. Das alte Element an Position *loc* wird gelöscht.

SetData `void SetData(int loc, const T& t)`

Das gegebene Element *t* ersetzt das aktuelle Element an Position *loc*.

SqueezeEntry `void SqueezeEntry(unsigned loc)`

Entfernt das Element an Position *loc* und verkleinert das Array um ein Element. Die Elemente an den Indexpositionen ab (*loc* + 1) werden um

eine Position nach unten verschoben. Das alte Element an Position *loc* wird gelöscht.

ZeroBase

```
unsigned ZeroBase(int loc) const
```

Liefert die Position relativ zu *lowerbound* zurück (das entspricht dem Wert *loc - lowerbound*).

Operatoren

operator []

```
T * & operator [] (int loc)
T * & operator [] (int loc) const
```

Siehe *TMArrayAsVector*, Operator []

Template TMIArrayAsVectorIterator — arrays.h

Implementiert ein Iterator-Objekt, mit dem *TMIArrayAsVector*-Objekte durchlaufen werden können. Basiert auf *TMVectorIteratorImp*.

Konstruktoren

```
TMIArrayAsVectorIterator(const TMIArrayAsVector<T,Alloc> & a) :
TMIVectorIteratorImp<T,Alloc>(a.data)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIArrayAsVector*-Objekte durchlaufen werden.

Public-Elementfunktionen

Current

```
T *Current();
```

Siehe *TMArrayAsVectorIterator::Current*

Restart

```
void Restart();
void Restart(unsigned start, unsigned stop);
```

Siehe *TMArrayAsVectorIterator::Restart*

Operatoren

operator ++

```
T *operator ++(int);
T *operator ++();
```

Siehe *TMArrayAsVectorIterator::++*

Template TIArrayAsVector arrays.h

TIArrayAsVector implementiert auf Basis eines Vektors ein indirektes Array von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Die Elementfunktionen und Operatoren finden Sie unter *TMIArrayAsVector*.

Konstruktoren

```
TIArrayAsVector(int upper, int lower =0, int delta = 0)
```

Erzeugt ein Array mit der oberen Grenze *upper*, der unteren Grenze *lower* und einem Größenzuwachs von *delta*.

Template TIArrayAsVectorIterator arrays.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TIArrayAsVector*-Objekten. Zur Speicherverwaltung wird *TStandardAllocator* benutzt. Die Elementfunktionen und Funktionen finden Sie unter *TMIArrayAsVectorIterator*.

Konstruktoren

```
TIArrayAsVectorIterator(const TIArrayAsVector<T> & a) :
TMIArrayAsVectorIterator<T,TStandardAllocator>(a)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TIArrayAsVector*-Objekten.

Template TMSArrayAsVector arrays.h

TMSArrayAsVector implementiert auf Basis eines Vektors ein sortiertes Array von Objekten des Typs *T*. Mit Ausnahme der Elementfunktion *AddAt* erbt *TMSArrayAsVector* alle Elementfunktionen und Operatoren von *TMArrayAsVector*. Siehe *TMArrayAsVector*.

Konstruktoren

```
TMSArrayAsVector(int upper, int lower =0, int delta = 0)
```

Erzeugt ein Array mit der oberen Grenze *upper*, der unteren Grenze *lower* und einem Größenzuwachs von *delta*. Für den Typ *T* muß der Operator < definiert sein.

Template TMSArrayAsVectorIterator arrays.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TMSArrayAs-Vector*-Objekten. Siehe *TMArrayAsVectorIterator*.

Konstruktoren

```
TMSArrayAsVectorIterator(const TMSArrayAsVector<T> & a)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TMSArrayAsVector*-Objekten.

Template TSArrayAsVector arrays.h

Implementiert auf Basis eines Vektors ein sortiertes Array von Objekten des Typs *T*. Mit Ausnahme der Elementfunktion *AddAt* erbt *TMSArrayAs-Vector* alle Elementfunktionen und Operatoren von *TMArrayAsVector*. Siehe *TMArrayAsVector*.

Konstruktoren

```
TSArrayAsVector(int upper, int lower =0, int delta = 0)
```

Erzeugt ein Array mit der oberen Grenze *upper*, der unteren Grenze *lower* und einem Größenzuwachs von *delta*. Für den Typ *T* ist ein < Operator erforderlich.

Template TSArrayAsVectorIterator arrays.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TSArrayAsVector*-Objekten. Siehe *TMArrayAsVectorIterator*.

Konstruktoren

```
TSArrayAsVectorIterator(const TSArrayAsVector<T> & a)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TSArrayAsVector*-Objekten.

Template TISArrayAsVector arrays.h

Implementiert auf Basis eines Vektors ein indirektes sortiertes Array von Objekten des Typs *T*. Die Elementfunktionen und Operatoren finden Sie unter *TMIArrayAsVector*.

Konstruktoren

```
TISArrayAsVector(int upper, int lower =0, int delta = 0)
```

Erzeugt ein indirektes Array mit der oberen Grenze *upper*, der unteren Grenze *lower* und einem Größenzuwachs von *delta*.

Template TISArrayAsVectorIterator arrays.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TISArrayAs-Vector*-Objekten. Siehe *TMArrayAsVectorIterator*.

Konstruktoren

```
TISArrayAsVectorIterator(const TISArrayAsVector<T> &a)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TISArrayAsVector*-Objekten.

Template TMISArrayAsVector arrays.h

Implementiert auf Basis eines Vektors ein indirektes sortiertes Array mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Elementfunktionen und Operatoren finden Sie unter *TMIArrayAsVector*.

Konstruktoren

```
TMISArrayAsVector(int upper, int lower = 0, int delta = 0)
```

Erzeugt ein indirektes Array mit der oberen Grenze *upper*, der unteren Grenze *lower* und einem Größenzuwachs von *delta*.

Template TMDDAssociation assoc.h

Implementiert eine Association mit eigener Speicherverwaltung, indem ein direkter Schlüssel (*K*) mit einem direktem Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K* über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

```
unsigned HashValue(K &);
```

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

```
TMDDAssociation()
```

Der Standardkonstruktor.

```
TMDDAssociation(const K &k, const V &v)
```

Erzeugt ein Objekt, das eine Kopie des Schlüsselobjekt *k* einer Kopie des Wertobjekts *v* zuordnet.

Public-Elementfunktionen

HashValue

```
unsigned HashValue()
```

Liefert den Hash-Wert der Schlüssels zurück.

Key

```
K Key()
```

Liefert den Schlüssel zurück.

Value

```
V Value()
```

Liefert eine Kopie der Daten zurück.

Operatoren

operator ==

```
int operator == (const TMDDAssociation<K,V,A> &a)
```

Testet zwei Schlüssel auf Gleichheit.

Template TDDAssociation assoc.h

Implementiert eine Association, indem ein direkter Schlüssel (*K*) mit einem direktem Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K* über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

```
unsigned HashValue(K &);
```

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Eine Beschreibung der Klassenelemente finden Sie unter *TMDDAssociation*.

Konstruktoren

`TDDAssociation()`

Der Standardkonstruktor.

`TDDAssociation(const K &k, const V &v)`

Erzeugt ein Objekt, das das Schlüsselobjekt *k* dem Wertobjekt *v* zuordnet.

Template TMDIAssociation — assoc.h

Implementiert eine Association mit eigener Speicherverwaltung, indem ein direkter Schlüssel (*K*) mit einem indirektem Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K* über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

```
unsigned HashValue(K &);
```

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

`TMDIAssociation()`

Der Standardkonstruktor.

`TMDIAssociation(K k, V *v)`

Erzeugt ein Objekt, das das Schlüsselobjekt *k* dem Wertobjekt **v* zuordnet.

Public-Elementfunktionen

HashValue

`unsigned HashValue()`

Liefert den Hash-Wert des Schlüssels zurück.

Key

`K Key()`

Liefert den Schlüssel zurück.

Value

`V * Value()`

Liefert einen Zeiger auf die Daten zurück.

Operatoren

operator ==

```
int operator == (const TMDIAssociation<K,V,A> &a)
```

Testet zwei Schlüssel auf Gleichheit.

Template TDIAssociation — assoc.h

Implementiert eine Association, indem ein direkter Schlüssel (*K*) mit einem indirekten Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K* über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

```
unsigned HashValue(K &);
```

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Eine Beschreibung der Klassenelemente finden Sie unter *TMDIAssociation*.

Konstruktoren

```
TDIAssociation()
```

Der Standardkonstruktor.

```
TDIAssociation(K k, V *v)
```

Erzeugt ein Objekt, das das Schlüsselobjekt *k* dem Wertobjekt **v* zuordnet.

Template TMIDAssociation — assoc.h

Implementiert eine Association mit eigener Speicherverwaltung, indem ein indirekter Schlüssel (*K*) mit einem direktem Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K* über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

```
unsigned HashValue(K &);
```

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

```
TMIDAssociation()
```

Der Standardkonstruktor.

`TMIDAssociation(K *k, V v)`

Erzeugt eine Objekt, das das Schlüsselobjekt **k* dem Wertobjekt *v* zuordnet.

Public-Elementfunktionen

HashValue `unsigned HashValue()`

Liefert den Hash-Wert des Schlüssels zurück.

Key `K * Key()`

Liefert einen Zeiger auf den Schlüssel zurück.

Value `V Value()`

Liefert eine Kopie der Daten zurück.

Operatoren

operator == `int operator == (const TMIDAssociation<K,V,A> &a)`

Testet zwei Schlüssel auf Gleichheit.

Template TIDAssociation — assoc.h

Implementiert eine Association, indem ein indirekter Schlüssel (*K*) mit einem direktem Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K* über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

`unsigned HashValue(K &);`

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Eine Beschreibung der Klassenelemente finden Sie unter *TMIDAssociation*.

Konstruktoren

`TIDAssociation()`

Der Standardkonstruktor.

`TIDAssociation(K *k, V v)`

Erzeugt ein Objekt, das das Schlüsselobjekt **k* dem Wertobjekt *v* zuordnet.

Template TMIIAssociation assoc.h

Implementiert eine Association mit eigener Speicherverwaltung, indem ein indirekter Schlüssel (*K*) mit einem indirektem Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K* über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

```
unsigned HashValue(K &);
```

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

`TMIIAssociation()`
Der Standardkonstruktor.

`TMIIAssociation(K *k, V *v)`
Erzeugt eine Objekt, das das Schlüsselobjekt **k* dem Wertobjekt **v* zuordnet.

Public-Elementfunktionen

HashValue

`unsigned HashValue()`
Liefert den Hash-Wert des Schlüssels zurück.

Key

`K *Key()`
Liefert einen Zeiger auf den Schlüssel zurück.

Value

`V *Value()`
Liefert einen Zeiger auf die Daten zurück.

Operatoren

operator ==

`int operator == (const TMIIAssociation<K,V,A> &a)`
Testet zwei Schlüssel auf Gleichheit.

Template TIIAssociation assoc.h

Implementiert eine Association, indem ein indirekter Schlüssel (*K*) mit einem indirektem Wert (*V*) verbunden wird. Es wird vorausgesetzt, daß *K*

über die Elementfunktion *HashValue* verfügt, oder daß eine globale Funktion mit folgendem Prototypen existiert:

```
unsigned HashValue(K &);
```

Die Klasse *K* muß zudem einen gültigen == Operator besitzen. Eine Beschreibung der Klassenelemente finden Sie unter *TMIIAssociation*.

Konstruktoren

```
TIIAssociation()
```

Der Standardkonstruktor.

```
TIIAssociation(K *k, V *v)
```

Erzeugt ein Objekt, das das Schlüsselobjekt **k* dem Wertobjekt **v* zuordnet.

Template TMBagAsVector — bags.h

Implementiert auf Basis eines Vektors einen Bag mit eigener Speicherverwaltung von Objekten des Typs *T*. Bags können im Gegensatz zu Sets doppelte Elemente enthalten.

Konstruktoren

```
TMBagAsVector(unsigned sz = DEFAULT_BAG_SIZE)
```

Erzeugt einen leeren Bag mit eigener Speicherverwaltung. *sz* enthält die Anzahl Einträge, die der Bag aufnehmen kann.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef int ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Public-Elementfunktionen

Add

`int Add(const T& t)`

Fügt das angegebene Objekt dem Bag hinzu.

Detach

```
int Detach(const T& t, TShouldDelete::DeleteType dt =
         TShouldDelete::NoDelete)
```

Entfernt das angegebene Objekt aus dem Bag. Der Wert von *dt* sowie der aktuelle Besitzerstatus bestimmen, ob das Objekt selbst gelöscht wird. *DeleteType* ist in der Basisklasse *TShouldDelete* als `enum {NoDelete, DefDelete, Delete}` definiert. Der Standardwert von *dt* ist *NoDelete* und bewirkt, daß das Objekt unabhängig vom Besitzer nicht gelöscht wird. Wird *dt* auf *Delete* gesetzt, so wird das Objekt unabhängig vom Besitzer in jedem Fall gelöscht. Enthält *dt* den Wert *DefDelete*, so wird das Objekt nur dann gelöscht, wenn der Bag Besitzer seiner Elemente ist.

Siehe auch: *TShouldDelete::ownsElemets*

FindMember

`T *FindMember(const T& t) const`

Liefert einen Zeiger auf das angegebene Objekt zurück, sofern es gefunden werden konnte. Andernfalls wird 0 zurückgegeben.

Flush

`void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)`

Entfernt alle Elemente aus dem Bag, ohne jedoch den Bag zu löschen. Der Wert von *dt* bestimmt, ob die Elemente selbst gelöscht werden. Standardmäßig wird das »Schicksal« der Elemente vom Besitzerstatus bestimmt (siehe Erläuterungen zur Elementfunktion *Detach*). Sie können *dt* auch auf *Delete* oder *Nodelete* setzen.

Siehe auch: *Detach*

ForEach

`void ForEach(void (*f)(T &, void *), void *args)`

ForEach erzeugt einen internen Iterator, mit dem die angegebene Funktion auf jedes Bag-Element angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

`unsigned GetItemsInContainer() const`

Liefert die Anzahl der Elemente im Bag zurück.

HasMember

`int HasMember(const T& t) const`

Liefert, wenn das gegebene Objekt im Bag gefunden werden konnte, den Wert 1, andernfalls den Wert 0 zurück.

IsEmpty

`int IsEmpty() const`

Liefert den Wert 1 zurück, falls der Bag kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 0 zurück.

Siehe auch: *FirstThat*, *ForEach*

Protected-Elementfunktionen

Find `virtual T *Find(const T&) const`

Liefert einen Zeiger auf das angegebene Objekt zurück, wenn es gefunden werden konnte. Andernfalls wird 0 zurückgeliefert.

Template TMBagAsVectorIterator bags.h

Implementiert ein Iterator-Objekt, mit dem *TMBagAsVector*-Objekte durchlaufen werden können. Eine Beschreibung der Elementfunktionen und Operatoren finden Sie unter *TMArrayAsVectorIterator*.

Konstruktoren

`TMBagAsVectorIterator(const TMBagAsVector<T,Alloc> & b)`

Erzeugt ein Iterator-Objekt, mit dem *TMBagAsVector*-Objekte durchlaufen werden.

Template TBagAsVector bags.h

Implementiert auf Basis eines Vektors einen Bag von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Klassenelemente finden Sie unter *TMBagAsVector*.

Konstruktoren

`TBagAsVector(unsigned sz = DEFAULT_BAG_SIZE)`

Erzeugt einen leeren Bag. *sz* enthält die Anzahl der Einträge, die der Bag aufnehmen kann.

Template TBagAsVectorIterator bags.h

Implementiert ein Iterator-Objekt, mit dem *TBagAsVector*-Objekte durchlaufen werden können. *TStandardAllocator* wird zur Speicherverwaltung

benutzt. Eine Beschreibung der Elementfunktionen und Operatoren finden Sie unter *TMArrayAsVectorIterator*.

Konstruktoren

`TBagAsVectorIterator(const TBagAsVector<T> & b)`

Erzeugt ein Iterator-Objekt, mit dem *TBagAsVector*-Objekte durchlaufen werden.

Template TMIBagAsVector — bags.h

Implementiert auf Basis eines Vektors einen Bag mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*.

Typdefinitionen

CondFunc `typedef int ( *CondFunc)(const T &, void *);`

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc `typedef int ( *IterFunc)(T &, void *);`

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

`TMIBagAsVector(unsigned sz = DEFAULT_BAG_SIZE)`

Erzeugt einen leeren, indirekten Bag mit eigener Speicherverwaltung. *sz* enthält die Anzahl Einträge, die der Bag aufnehmen kann.

Public-Elementfunktionen

Add `int Add(T *t)`

Fügt den angegebenen Objektzeiger dem Bag hinzu.

Detach `int Detach(T *t, DeleteType dt = NoDelete)`

Entfernt den angegebenen Objektzeiger aus dem Bag. Der Wert von *dt* sowie der aktuelle Besitzerstatus bestimmen, ob das Objekt selbst gelöscht wird. *DeleteType* ist in der Basisklasse *TShouldDelete* als `enum {NoDelete, DefDelete, Delete}` definiert. Der Standardwert von *dt* ist NoDe-

lete und bewirkt, daß das Objekt unabhängig vom Besitzer nicht gelöscht wird. Wird *dt* auf *Delete* gesetzt, so wird das Objekt unabhängig vom Besitzer in jedem Fall gelöscht. Enthält *dt* den Wert *DefDelete*, so wird das Objekt nur dann gelöscht, wenn der Bag Besitzer seiner Elemente ist.

Siehe auch: *TShouldDelete::ownsElemets*

FindMember `T *FindMember(T *t) const`

Liefert einen Zeiger auf das angegebene Objekt zurück, sofern es gefunden werden konnte. Andernfalls wird 0 zurückgegeben.

FirstThat `T *FirstThat(CondFunc f, void *args) const`

Siehe *TMBagAsVector::FirstThat*

Flush `void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)`

Entfernt alle Elemente aus dem Bag, ohne jedoch den Bag zu löschen. Der Wert von *dt* bestimmt, ob die Elemente selbst gelöscht werden. Standardmäßig wird das »Schicksal« der Elemente vom Besitzerstatus bestimmt (siehe Erläuterungen zur Elementfunktion *Detach*). Sie können *dt* auch auf *Delete* oder *Nodelete* setzen.

Siehe auch: *Detach*

ForEach `void ForEach(IterFunc f, void *args)`

Siehe *TMBagAsVector::ForEach*

GetItemsInContainer `unsigned GetItemsInContainer() const`

Liefert die Anzahl der Elemente im Bag zurück.

HasMember `int HasMember(const T& t) const`

Liefert, wenn das gegebene Objekt im Bag gefunden werden konnte, den Wert 1, andernfalls den Wert 0 zurück.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls der Bag kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 0 zurück.

LastThat `T *LastThat(CondFunc f, void *args) const`

Siehe *TMBagAsVector::LastThat*

Template TMIBagAsVectorIterator — bags.h

Implementiert ein Iterator-Objekt, mit dem *TMIBagAsVector*-Objekte durchlaufen werden können. Eine Beschreibung der Elementfunktionen und Operatoren finden Sie unter *TMArrayAsVectorIterator*.

Konstruktoren

`TMIBagAsVectorIterator(const TMIBagAsVector<T,Alloc> & s)`

Erzeugt ein Iterator-Objekt, mit dem *TMIBagAsVector*-Objekte durchlaufen werden.

Template TIBagAsVector bags.h

Implementiert auf Basis eines Vektors einen Bag von Zeigern auf Objekte des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Elementfunktionen und Operatoren finden Sie unter *TMIBagAsVector*.

Konstruktoren

`TIBagAsVector(unsigned sz = DEFAULT_BAG_SIZE)`

Siehe Konstruktor von *TMIBagAsVector*

Template TIBagAsVectorIterator bags.h

Implementiert ein Iterator-Objekt, mit dem *TIBagAsVector*-Objekte durchlaufen werden können. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Elementfunktionen und Operatoren finden Sie unter *TMArrayAsVectorIterator*.

Konstruktoren

`TIBagAsVectorIterator(const TIBagAsVector<T> & s)`

Siehe Konstruktor von *TMIBagAsVectorIterator*

Template TBinarySearchTreeImp binimp.h

Implementiert einen unbalancierten Binärbaum. Die Klasse *T* muß über die Operatoren < und == sowie einen Standardkonstruktor verfügen.

Public-Elementfunktionen

Add

`int Add(const T& t)`

Erzeugt einen neuen Binärbaumknoten und fügt eine Kopie des Objekts *t* dort ein.

Detach

`int Detach(T t, int del = 0)`

Entfernt den Knoten mit dem Objekt *t* aus dem Baum.

Find

`T *Find(const T& t) const`

Liefert einen Zeiger auf den Knoten zurück, der *t* enthält.

Flush

`void Flush(int del = 0);`

Entfernt alle Einträge aus dem Baum.

ForEach

`void ForEach(IterFunc f, void *args, IteratorOrder order = InOrder)`

Erzeugt einen internen Iterator, der die durch *f* angegebene Funktion für jedes Baumelement ausführt. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

`unsigned GetItemsInContainer()`

Liefert die Anzahl der Elemente im Baum zurück.

Parent::IsEmpty

`int IsEmpty`

Liefert bei leerem Baum den Wert 1, ansonsten den Wert 0 zurück.

Protected-Elementfunktionen

EqualTo

`virtual int EqualTo(BinNode *n1, BinNode *n2)`

Überprüft die Gleichheit zweier Knoten.

LessThan

`virtual int LessThan(BinNode *n1, BinNode *n2)`

Überprüft, ob Knoten *n1* kleiner ist als Knoten *n2*.

DeleteNode

`virtual void DeleteNode(BinNode *node, int del)`

Löscht den Knoten *node*. Der zweite Parameter wird ignoriert.

Template TBinarySearchTreeIteratorImp — binimp.h

Implementiert einen Iterator, mit dem *TBinarySearchTreeImp*-Objekte durchlaufen werden können.

Konstruktoren

```
TBinarySearchTreeIteratorImp(TBinarySearchTreeImp<T>& tree,
TBinarySearchTreeBase::IteratorOrder order =
TBinarySearchTreeBase::InOrder) : TBinaryTreeExternalIteratorBase(tree,
order), CurNode(static_cast<TBinaryNodeImp<T>*>(Next()))
```

Erzeugt ein Iterator-Objekt mit dem *TBinarySearchTreeImp*-Container durchlaufen werden können.

Public-Elementfunktionen

Current

```
const T& Current() const
```

Liefert das aktuelle Objekt zurück.

Restart

```
void Restart()
```

Startet eine neue Iteration, beginnend mit dem Baumanfang.

Operatoren

operator int

```
operator int() const
```

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++

```
const T& operator ++(int)
```

Bewegt den Iterator auf das nächste Objekt im Baum und liefert das zuvor aktuelle Objekt zurück (Post-Inkrement).

```
const T& operator ++()
```

Bewegt den Iterator auf das nächste Objekt im Baum und liefert das dann aktuelle Objekt zurück (Prä-Inkrement).

Template TIBinarySearchTreeImp binimp.h

Implementiert einen indirekten, nicht balancierten Binärbaum. Die Klasse *T* muß über die Operatoren < und == sowie einen Standardkonstruk-tor verfügen.

Public-Elementfunktionen

Add

```
int Add(T *t)
```

Erzeugt einen neuen Binärbaumknoten und fügt den Zeiger auf das Objekt *t* dort ein.

Detach

```
int Detach(T *t, int del = 0)
```

Entfernt den Knoten aus dem Baum, der das Element *t* enthält. Ist *del* gleich 1, so wird das Element gelöscht.

Find

```
T *Find(T t) const
```

Liefert einen Zeiger auf den Knoten zurück, der **t* enthält.

Flush

```
void Flush(int del = 0)
```

Entfernt alle Elemente aus dem Baum. Die Elemente selbst werden gelöscht, wenn *del* den Wert 1 hat. Ist *del* gleich 0, so werden die Elemente selbst nicht gelöscht.

ForEach

```
void ForEach( void (*func)(T &, void *), void *args,
IteratorOrder order = InOrder)
```

Erzeugt einen internen Iterator, der die durch *f* angegebene Funktion für jedes Baumelement ausführt. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

```
unsigned GetItemsInContainer()
```

Liefert die Anzahl der Elemente im Baum zurück.

Parent::IsEmpty

```
int IsEmpty
```

Liefert bei leerem Baum den Wert 1, ansonsten den Wert 0 zurück.

Protected-Elementfunktionen

EqualTo

```
virtual int EqualTo(BinNode *n1, BinNode *n2)
```

Überprüft die Gleichheit zweier Knoten.

LessThan

```
virtual int LessThan(BinNode *n1, BinNode *n2)
```

Überprüft, ob Knoten *n1* kleiner ist als Knoten *n2*.

DeleteNode

```
virtual void DeleteNode(BinNode *node, int del)
```

Löscht den Knoten *node*. Der zweite Parameter wird ignoriert.

Template TIBinarySearchTreeIteratorImp binimp.h

Implementiert einen Iterator, mit dem *TIBinarySearchTreeImp*-Objekte durchlaufen werden können.

Konstruktoren

```
TIBinarySearchTreeIteratorImp(TIBinarySearchTreeImp<T>& tree,
TBinarySearchTreeBase::IteratorOrder order =
TBinarySearchTreeBase::InOrder) :
TBinarySearchTreeIteratorImp(TVoidPointer(tree,order)
```

Erzeugt ein Iterator-Objekt mit dem *TIBinarySearchTreeImp*-Container durchlaufen werden können.

Public-Elementfunktionen

Current

```
T *Current() const
```

Liefert einen Zeiger auf das aktuelle Objekt zurück.

Restart

```
void Restart()
```

Startet eine neue Iteration, beginnend mit dem Baumanfang.

Operatoren

operator int

```
operator int() const
```

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++

```
T *operator ++(int)
```

Bewegt den Iterator auf das nächste Objekt im Baum und liefert das zuvor aktuelle Objekt zurück (Post-Inkrement).

```
T *operator ++()
```

Bewegt den Iterator auf das nächste Objekt im Baum und liefert das dann aktuelle Objekt zurück (Prä-Inkrement).

Template TMDequeAsVector — deques.h

Implementiert auf Basis eines Vektors eine Dequeue (Warteschlange mit zwei Enden) mit eigener Speicherverwaltung von Objekten des Typs *T*.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef int ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

```
TMDequeAsVector(unsigned max = DEFAULT_DEQUE_SIZE)
```

Erzeugt eine Dequeue der Größe *max*.

Public-Elementfunktionen

FirstThat

```
T *FirstThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das erste Dequeue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush

```
void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)
```

Entfernt alle Elemente aus der Dequeue, ohne jedoch die Dequeue zu löschen. Der Wert von *dt* bestimmt, ob die Elemente selbst gelöscht werden. Standardmäßig wird das »Schicksal« der Elemente vom Besitzerstatus bestimmt.

Siehe auch: *TShouldDelete::ownsElements*

ForEach

```
void ForEach(IterFunc f, void *args)
```

Wendet die Funktion *f* auf alle Dequeue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

GetItemsInContainer

```
unsigned GetItemsInContainer() const
```

Liefert die Anzahl der Elemente in der Dequeue zurück.

GetLeft

```
T GetLeft()
```

Liefert das Objekt am linken Ende der Dequeue zurück und entfernt es aus der Dequeue. Bei leerer Dequeue löst die Debug-Version eine Exception aus.

Siehe auch: *PeekLeft*

GetRight

`T GetRight()`

Liefert das Objekt am rechten Ende der Dequeue zurück und entfernt es aus der Dequeue. Bei leerer Dequeue wird 0 zurückgeliefert.

Siehe auch: *PeekRight*

IsEmpty

`int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Dequeue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull

`int IsFull() const`

Liefert den Wert 1 zurück, falls die Dequeue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat

`T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Objekt der Dequeue zurück, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

PeekLeft

`const T& PeekLeft() const`

Liefert das Objekt am linken Ende der Dequeue. Das Objekt bleibt dabei in der Dequeue.

Siehe auch: *GetLeft*

`const T& PeekRight() const`

Liefert das Objekt am rechten Ende der Dequeue. Das Objekt bleibt dabei in der Dequeue.

Siehe auch: *GetRight*

PutLeft

`void PutLeft(const T&)`

Fügt das gegebene Objekt am linken Ende der Dequeue ein.

PutRight

`void PutRight(const T&)`

Fügt das gegebene Objekt am rechten Ende der Dequeue ein.

Protected-Datenelemente

Data

`Vect Data;`

Der Vektor, der die Daten der Dequeue enthält.

Left

`unsigned Left;`

Indexposition des Elements am linken Ende der Dequeue.

Right

`unsigned Right;`

Indexposition des Elements am rechten Ende der Dequeue.

Protected-Elementfunktionen

Next

```
unsigned Next(unsigned index) const
```

Liefert (*index* + 1) zurück. Bei Überschreiten des rechten Endes wird automatisch mit dem linken Ende der Dequeue fortgefahren.

Prev

```
unsigned Prev(unsigned index) const
```

Liefert (*index* - 1) zurück. Bei Überschreiten des linken Endes wird automatisch mit dem rechten Ende der Dequeue fortgefahren.

Template TMDequeAsVectorIterator deques.h

Implementiert ein Iterator-Objekt für eine auf Vektoren basierende Dequeue mit eigener Speicherverwaltung.

Konstruktoren

```
TMDequeAsVectorIterator(const TMDequeAsVector<T,Alloc> &d)
```

Erzeugt ein Objekt zum Durchlaufen von *TMDequeAsVector*-Objekten.

Public-Elementfunktionen

Current

```
const T& Current();
```

Liefert das aktuelle Objekt zurück.

Restart

```
void Restart();
```

Startet die Iteration erneut.

Operatoren

operator ++

```
const T& operator ++ (int);
const T& operator ++();
```

Siehe *TMArrayAsVectorIterator*, Operator ++

operator int

```
operator int();
```

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

Template TDequeAsVector deques.h

Implementiert auf Basis eines Vektors eine Dequeue (Warteschlange mit zwei Enden) von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Klassenelemente finden Sie unter *TMDequeAsVector*.

Konstruktoren

```
TDequeAsVector(unsigned max = DEFAULT_DEQUE_SIZE)
```

Erzeugt eine Dequeue der Größe *max*.

Template TDequeAsVectorIterator deques.h

Implementiert ein Iterator-Objekt für eine auf Vektoren basierende Dequeue. Eine Beschreibung der Klassenelemente finden Sie unter *TMDequeAsVectorIterator*

Konstruktoren

```
TDequeAsVectorIterator(const TDequeAsVector<T> &d)
```

Erzeugt ein Objekt zum Durchlaufen von *TDequeAsVector*-Objekten.

Template TMIDequeAsVector deques.h

Implementiert auf Basis eines Vektors eine indirekte Dequeue (Warteschlange mit zwei Enden) mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef int ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

`TMIDequeAsVector(unsigned max = DEFAULT_DEQUE_SIZE)`

Erzeugt eine indirekte Dequeue der Größe *max*.

Public-Elementfunktionen

FirstThat

`T *FirstThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das erste Dequeue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush

`void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)`

Siehe *TMDequeAsVectorImp::Flush*

ForEach

`void ForEach(IterFunc f, void *args)`

Wendet die Funktion *f* auf alle Dequeue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

GetItemsInContainer

`unsigned GetItemsInContainer() const`

Liefert die Anzahl der Elemente in der Dequeue zurück.

GetLeft

`T *GetLeft()`

Liefert den Zeiger auf das Objekt am linken Ende der Dequeue zurück und entfernt ihn aus der Dequeue. Bei leerer Dequeue wird 0 zurückgeliefert.

Siehe auch: *PeekLeft*

GetRight

`T *GetRight()`

Liefert den Zeiger auf das Objekt am rechten Ende der Dequeue zurück und entfernt ihn aus der Dequeue. Bei leerer Dequeue wird 0 zurückgeliefert.

Siehe auch: *PeekRight*

IsEmpty

`int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Dequeue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull

`int IsFull() const`

Liefert den Wert 1 zurück, falls die Dequeue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat

`T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Objekt der Dequeue zurück, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

PeekLeft `T *PeekLeft() const`

Liefert das Objekt am linken Ende der Dequeue. Das Objekt bleibt dabei in der Dequeue.

Siehe auch: *GetLeft*

PeekRight `T *PeekRight() const`

Liefert das Objekt am rechten Ende der Dequeue. Das Objekt bleibt dabei in der Dequeue.

Siehe auch: *GetRight*

PutLeft `void PutLeft(T *t)`

Fügt das gegebene Objekt am linken Ende der Dequeue ein.

PutRight `void PutRight(T *t)`

Fügt das gegebene Objekt am rechten Ende der Dequeue ein.

Template TMIDequeAsVectorIterator deques.h

Implementiert ein Iterator-Objekt für die Gruppe der auf Vektoren basierenden indirekten Dequeues mit eigener Speicherverwaltung. Siehe *TMDequeAsVectorIterator*.

Konstruktoren

```
TMIDequeAsVectorIterator(const TMIDequeAsVector<T,Alloc> &d)
```

Erzeugt ein Objekt zum Durchlaufen von *TMIDequeAsVector*-Objekten.

Template TIDequeAsVector deques.h

Implementiert auf Basis eines Vektors eine indirekte Dequeue (Warteschlange mit zwei Enden) von Zeigern auf Objekte des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Klassenelemente finden Sie unter *TMIDequeAsVector*.

Konstruktoren

```
TIDequeAsVector(unsigned max = DEFAULT_DEQUE_SIZE) :
TMIDequeAsVector<T,TStandardAllocator>(sz)
```

Erzeugt eine indirekte Dequeue der Größe *max*.

Template TIDequeAsVectorIterator deques.h

Implementiert ein Iterator-Objekt für die Gruppe der auf Vektoren basierenden indirekten Dequeues. Siehe *TMDequeAsVectorIterator*.

Konstruktoren

`TIDequeAsVectorIterator(const TIDequeAsVector<T> &d)`

Erzeugt ein Objekt zum Durchlaufen von *TIDequeAsVector*-Objekten.

Template TMDequeAsDoubleList deques.h

Implementiert auf Basis einer doppelt verketteten Liste eine Dequeue (Warteschlange mit zwei Enden) mit eigener Speicherverwaltung von Objekten des Typs *T*.

Typdefinitionen

CondFunc `typedef int ( *CondFunc)(const T &, void *);`

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc `typedef int ( *IterFunc)(T &, void *);`

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Erzeugt eine Dequeue der Größe *max*.

Public-Elementfunktionen

FirstThat `T *FirstThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das erste Dequeue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush `void Flush(int del)`

Entfernt alle Elemente aus der Dequeue. Siehe *TMDequeAsVector::Flush*.

ForEach `void ForEach(IterFunc f, void *args)`

Wendet die Funktion *f* auf alle Dequeue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

GetItemsInContainer `unsigned GetItemsInContainer() const`

Liefert *itemsInContainer* zurück.

GetLeft `T GetLeft()`

Liefert das Objekt am linken Ende der Dequeue zurück und entfernt es aus der Dequeue.

GetRight `T GetRight()`

Liefert das Objekt am rechten Ende der Dequeue zurück und entfernt es aus der Dequeue.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Dequeue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 1 zurück, falls die Dequeue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *), void *args) const`

Liefert einen Zeiger auf das letzte Objekt der Dequeue zurück, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

PeekLeft `const T& PeekLeft() const`

Liefert eine Referenz auf das Objekt am linken Ende der Dequeue. Das Objekt bleibt dabei in der Dequeue.

Siehe auch: *GetLeft*

PeekRight `const T& PeekRight() const`

Liefert eine Referenz auf das Objekt am rechten Ende der Dequeue. Das Objekt bleibt dabei in der Dequeue.

Siehe auch: *GetRight*

PutLeft `void PutLeft(const T&)`

Fügt das gegebene Objekt am linken Ende der Dequeue ein.

PutRight `void PutRight(const T&)`

Fügt das gegebene Objekt am rechten Ende der Dequeue ein.

Template TMDequeAsDoubleListIterator — deques.h

Implementiert ein Iterator-Objekt für eine auf einer doppelt verketteten Listen basierende Dequeue. Siehe *TMDoubleListIteratorImp*.

Konstruktoren

`TMDequeAsDoubleListIterator(const TMDequeAsDoubleList<T, Alloc> &s)`

Erzeugt ein Objekt zum Durchlaufen von *TDequeAsDoubleList*-Objekten.

Template TDequeAsDoubleList deques.h

Implementiert auf Basis einer doppelt verketteten Liste eine Dequeue (Warteschlange mit zwei Enden) von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Die Elemente finden Sie unter *TMDequeAsDoubleList*.

Template TDequeAsDoubleListIterator deques.h

Implementiert ein Iterator-Objekt für eine auf einer doppelt verketteten Listen basierende Dequeue.

Konstruktoren

`TDequeAsDoubleListIterator(const TDequeAsDoubleList<T> &s)`

Erzeugt ein Objekt zum Durchlaufen von *TDequeAsDoubleList*-Objekten.

Template TMIDequeAsDoubleList deques.h

Implementiert auf Basis einer doppelt verketteten Liste eine indirekte Dequeue (Warteschlange mit zwei Enden) mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*.

Typdefinitionen

CondFunc `typedef int ( *CondFunc)(const T &, void *);`

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc `typedef int ( *IterFunc)(T &, void *);`

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Erzeugt eine indirekte Dequeue der Größe *max*.

Public-Elementfunktionen

FirstThat `T *FirstThat(CondFunc f, void *), void *args) const`

Liefert einen Zeiger auf das erste Dequeue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush `void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)`

Entfernt alle Elemente aus der Dequeue, ohne sie jedoch zu löschen. Siehe *TMDequeAsVector::Flush*.

ForEach `void ForEach(IterFunc f, void *args)`

Wendet die Funktion *f* auf alle Dequeue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

GetItemsInContainer `unsigned GetItemsInContainer() const`

Liefert die Anzahl der Elemente in der Dequeue zurück.

GetLeft `T *GetLeft()`

Liefert den Zeiger auf das Objekt am linken Ende der Dequeue zurück und entfernt ihn aus der Dequeue. Bei leerer Dequeue wird 0 zurückgeliefert.

Siehe auch: *PeekLeft*

GetRight `T *GetRight()`

Liefert den Zeiger auf das Objekt am rechten Ende der Dequeue zurück und entfernt ihn aus der Dequeue. Bei leerer Dequeue wird 0 zurückgeliefert.

Siehe auch: *PeekRight*

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Dequeue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 1 zurück, falls die Dequeue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *), void *args) const`

Liefert einen Zeiger auf das letzte Objekt der Dequeue zurück, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

PeekLeft `T *PeekLeft() const`

Siehe *TMIDequeAsVector::PeeklLeft*

Siehe auch: *GetLeft*

PeekRight `T *PeekRight() const`

Siehe *TMIDequeAsVector::PeekRight*

Siehe auch: *GetRight*

PutLeft `void PutLeft(T *t)`

Siehe *TMIDequeAsVector::PutLeft*

PutRight `void PutRight(T *t)`

Siehe *TMIDequeAsVector::PutRight*

Template TMIDequeAsDoubleListIterator deques.h

Implementiert ein Iterator-Objekt für auf doppelt verketteten Listen basierende indirekte Dequeues mit eigener Speicherverwaltung. Siehe *TMDequeAsDoubleListIterator*.

Konstruktoren

`TMIDequeAsDoubleListIterator(const TMIDequeAsDoubleList<T, Alloc> &s)`

Erzeugt ein Objekt zum Durchlaufen von *TMIDequeAsDoubleList*-Objekten.

Template TIDequeAsDoubleList deques.h

Implementiert auf Basis einer doppelt verketteten Liste eine indirekte Dequeue (Warteschlange mit zwei Enden) von Zeigern auf Objekte des Typs *T*. Siehe *TMIDequeAsDoubleList*.

Template TIDequeAsDoubleListIterator deques.h

Implementiert ein Iterator-Objekt für auf doppelt verketteten Listen basierende indirekte Dequeues. Siehe *TDequeAsDoubleListIterator*.

Konstruktoren

`TIDequeAsDoubleListIterator(const TIDequeAsDoubleList<T> &s)`

Erzeugt ein Objekt zum Durchlaufen von *TIDequeAsDoubleList*-Objekten.

Implementiert auf Basis einer Hash-Tabelle ein Dictionary mit eigener Speicherverwaltung. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung. *T* muß eine der vier Association-Typen sein und neben einem Standardkonstruktor einen gültigen == Operator und eine sinnvolle Kopiersemantik besitzen.

Protected-Datenelemente

HashTable

`TMHashTableImp<T,A> HashTable;`

Implementiert die zugrundeliegende Hash-Tabelle.

Konstruktoren

`TMDictionaryAsHashTable(unsigned size = DEFAULT_HASH_TABLE_SIZE)`

Erzeugt ein Dictionary mit der angegebenen Größe.

Public-Elementfunktionen

Add

`int Add(const T&)`

Fügt das angegebene Objekt dem Dictionary hinzu.

Detach

`int Detach(const T& t, int del = 0)`

Entfernt das angegebene Objekt aus dem Dictionary. Ist *del* gleich 1, so wird das Objekt gelöscht. Bei *del* gleich 0 wird es nicht gelöscht.

Find

`T *Find(const T& t)`

Liefert einen Zeiger auf das Element *t* zurück.

Flush

`void Flush(int del = 0)`

Entfernt alle Elemente aus dem Dictionary. Ist *del* gleich 1, so werden die Elemente selbst gelöscht. Bei *del* gleich 0 werden sie nicht gelöscht.

ForEach

`void ForEach(void (*f)(T &, void *), void * args)`

ForEach erzeugt einen internen Iterator, mit dem die angegebene Funktion auf jedes Dictionary-Element angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

`inline unsigned GetItemsInContainer()`

Liefert die Anzahl der Elemente im Dictionary zurück.

IsEmpty

```
inline int IsEmpty()
```

Liefert den Wert 1 zurück, falls das Dictionary kein Element enthält. Andernfalls wird 0 zurückgeliefert.

Template TMDictionaryAsHashTableIterator — dict.h

Implementiert ein Iterator-Objekt für *TMDictionaryAsHashTable*-Objekte. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

```
TMDictionaryAsHashTableIterator(TMDictionaryAsHashTable<T,A> & t)
```

Erzeugt ein Objekt zum Durchlaufen von *TMDictionaryAsHashTable*-Objekten.

Public-Elementfunktionen

Current

```
T Current();
```

Liefert das aktuelle Objekt zurück.

Restart

```
void Restart();
```

Startet die Iteration erneut.

Operatoren

operator int

```
operator int();
```

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++

```
const T& operator ++(int);
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

```
const T& operator ++();
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

Template TDictionaryAsHashTable dict.h

Implementiert auf Basis einer doppelt verketteten Liste ein Dictionary von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. *T* muß einer der vier Association-Typen sein und neben einem Standardkonstruktor einen gültigen == Operator und eine sinnvolle Kopiersemantik besitzen. Eine Beschreibung der Klassenelemente finden Sie unter *TMDictionaryAsHashTable*.

Konstruktoren

```
TDictionaryAsHashTable(unsigned size = DEFAULT_HASH_TABLE_SIZE)
```

Erzeugt ein Dictionary der Größe *size*.

Template TDictionaryAsHashTableIterator dict.h

Implementiert ein Iterator-Objekt für *TDictionaryAsHashTable*-Objekte. *TStandardAllocator* wird zur Speicherverwaltung benutzt.

Konstruktoren

```
TDictionaryAsHashTableIterator(TDictionaryAsHashTable<T> &d)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TDictionaryAsHashTable*-Objekten.

Template TMIDictionaryAsHashTable dict.h

Implementiert auf Basis einer Hash-Tabelle ein indirektes Dictionary mit eigener Speicherverwaltung. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung. *T* muß eine *TAssociation*-Klasse sein.

Konstruktoren

```
TMIDictionaryAsHashTable(unsigned size = DEFAULT_HASH_TABLE_SIZE)
```

Erzeugt ein indirektes Dictionary mit der angegebenen Größe.

Public-Elementfunktionen

Add

`int Add(T *t)`

Fügt den angegebenen Objektzeiger dem Dictionary hinzu, sofern er noch nicht im Dictionary enthalten ist.

Detach

`int Detach(T *t, int del = 0)`

Entfernt den angegebenen Zeiger aus dem Dictionary. Ist *del* gleich 1, so wird das Element gelöscht. Bei *del* gleich 0 wird es nicht gelöscht.

Find

`T *Find(T *t)`

Liefert einen Zeiger auf das Element *t* zurück.

Flush

`void Flush(int del = 0)`

Entfernt alle Elemente aus dem Dictionary. Ist *del* gleich 1, so werden die Elemente selbst gelöscht. Bei *del* gleich 0 werden sie nicht gelöscht.

ForEach

`void ForEach(void (*f)(T &, void *), void * args)`

ForEach erzeugt einen internen Iterator, mit dem die durch *f* angegebene Funktion auf jedes Dictionary-Element angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

`inline unsigned GetItemsInContainer()`

Liefert die Anzahl der Elemente im Dictionary zurück.

IsEmpty

`inline int IsEmpty()`

Liefert den Wert 1 zurück, falls das Dictionary kein Element enthält. Andernfalls wird 0 zurückgeliefert.

Template TMIDictionaryAsHashTableIterator dict.h

Implementiert ein Iterator-Objekt für *TMIDictionaryAsHashTable*-Objekte. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

`TMIDictionaryAsHashTableIterator(TMIDictionaryAsHashTable<T,A> & t)`

Erzeugt ein Objekt zum Durchlaufen von *TMIDictionaryAsHashTable*-Objekten.

Public-Elementfunktionen

Current

`T *Current();`

Liefert den Zeiger auf das aktuelle Objekt zurück.

Restart

`void Restart();`

Startet die Iteration beginnend mit dem Dictionary-Anfang erneut.

Operatoren

operator int

`operator int();`

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++

`T *operator ++(int);`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert den Zeiger auf das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

`T *operator ++();`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert den Zeiger auf das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

Template TIDictionaryAsHashTable dict.h

Implementiert auf Basis einer doppelt verketteten Liste ein indirektes Dictionary von Zeigern auf Objekte des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. *T* muß eine der vier Association-Typen sein. Eine Beschreibung der Klassenelemente finden Sie unter *TMIDictionaryAsHashTable*.

Konstruktoren

`TIDictionaryAsHashTable(unsigned size = DEFAULT_HASH_TABLE_SIZE)`

Erzeugt ein indirektes Dictionary der Größe *size*.

Template TIDictionaryAsHashTableIterator dict.h

Implementiert ein Iterator-Objekt für *TIDictionaryAsHashTable*-Objekte. Eine Beschreibung der Klassenelemente finden Sie unter *TMIDictionaryAsHashTableIterator*.

Konstruktoren

```
TIDictionaryAsHashTableIterator(TIDictionaryAsHashTable<T> &d)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TIDictionaryAsHashTable*-Objekten.

Template TDictionary dict.h

Eine Kurzschreibweise für *TDictionaryAsHashTable*. Siehe *TDictionaryAsHashTable*.

Template TDictionaryIterator dict.h

Eine Kurzschreibweise für *TDictionaryAsHashTableIterator*. Siehe *TDictionaryAsHashTableIterator*.

Konstruktoren

```
TDictionaryIterator(TDictionary<T> & a)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TDictionary*-Objekten.

Template TMDoubleListElement dlistimp.h

Diese Klasse definiert die Knoten für die Klassen *TMDoubleListImp* und *TMIDoubleListImp*.

Public-Datenelemente

data

```
T data;
```

Datenobjekt, das in der doppelt verketteten Liste enthalten ist.

Next

```
TDoubleListElement<T> *Next;
```

Ein Zeiger auf das nächste Element in der doppelt verketteten Liste.

Prev

```
TDoubleListElement<T> *Prev;
```

Ein Zeiger auf das vorhergehende Element in der doppelt verketteten Liste.

Konstruktoren

```
TMDoubleListElement();
```

Erzeugt ein Element einer doppelt verketteten Liste.

```
TMDoubleListElement(T t, TMDoubleListElement<T> *p);
```

Erzeugt ein Element einer doppelt verketteten Liste und fügt es in die Liste hinter dem Objekt ein, auf das *p* zeigt.

Operatoren

operator delete

```
void operator delete(void *);
```

Löscht ein Objekt.

operator new

```
void *operator new(size_t sz);
```

Reserviert einen Speicherblock der Größe *sz* und liefert einen Zeiger auf den Speicherblock zurück.

Template TMDoubleListImp — dlistimp.h

Implementiert eine doppelt verkettete Liste mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef int ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

`TMDoubleListImp()`

Erzeugt eine leere doppelt verkettete Liste mit eigener Speicherverwaltung.

Public-Elementfunktionen

Add `int Add(const T&)`

Fügt das angegebene Objekt am Anfang der Liste ein.

AddAtTail `int AddAdTail(const T&)`

Fügt das angegebene Objekt am Ende der Liste ein.

Detach `int Detach(const T& , int = 0)`

Entfernt das erste Vorkommen des angegebenen Objekts, wobei ab Listenanfang gesucht wird. Der Integerparameter entscheidet bei indirekten Listen, ob das aus der Liste entfernte Element selbst gelöscht wird. Bei direkten Listen wird der zweite Parameter ignoriert. Einzelheiten finden Sie unter *TShouldDelete*.

FirstThat `T *FirstThat(int ( *f)(const T &, void *), void * args) const`

Liefert einen Zeiger auf das erste Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::FirstThat*.

Flush `void Flush(int = 0)`

Entfernt alle Elemente aus der Liste. Siehe *TMArrayAsVector::Flush*.

ForEach `void ForEach(IterFunc f, void *args)`

ForEach erzeugt einen internen Iterator, mit dem die durch *f* angegebene Funktion auf jedes Listenelement angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Liste kein Element enthält. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(int (*f)(const T &, void *), void *args) const`

Liefert einen Zeiger auf das letzte Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

PeekHead `const T& PeekHead() const`

Liefert Referenz auf das Datenelements *Head* zurück, ohne das Element aus der Liste zu entfernen.

PeekTail `const T& PeekTail() const`

Liefert eine Referenz auf das Datenelement *Tail* zurück, ohne das Element aus der Liste zu entfernen.

Protected-Datenelemente

Head, Tail `TMDoubleListElement<T> Head, Tail;`

Das erste und das letzte Listenelement.

Protected-Elementfunktionen

FindDetach `virtual TMDoubleListElement<T> *FindDetach(const T& t)`

Stellt fest, ob sich ein Element in der Liste befindet, und liefert einen Zeiger auf seine Vorgänger zurück. Wird das Element nicht gefunden, so liefert *FindDetach* den Wert 0 zurück.

FindPred `virtual TMDoubleListElement<T> *FindPred(const T&)`

Sucht das Listenelement, das vor dem übergebenen Element steht. Die Funktion überprüft nicht, ob sich der angegebene Parameter tatsächlich in der Liste befindet. *FindPred* ist beim Einfügen nützlich.

Template TMDoubleListIteratorImp — dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen direkten, doppelt verketteten Listen. Für indirekte Listen ist *TMIDoubleListIterator* vorgesehen.

Konstruktoren

`TMDoubleListIteratorImp(const TDoubleListImp<T> &l)`

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TDoubleListImp*-Objekten.

Public-Elementfunktionen

Current `const T& Current();`

Liefert das aktuelle Objekt zurück.

Restart `void Restart();`

Startet die Iteration beginnend mit dem Listenanfang erneut.

Operatoren

operator int

```
operator int();
```

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++

```
const T& operator ++(int);
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

```
const T& operator ++();
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

operator --

```
const T& operator --(int);
```

Verschiebt die Iterator-Position auf das vorherige Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Dekrement).

```
const T& operator --();
```

Verschiebt die Iterator-Position auf das vorherige Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Post-Dekrement).

Template TDoubleListImp — dlistimp.h

Implementiert eine doppelt verkettete Liste von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen. Siehe *TMDoubleListImp.*

Konstruktoren

```
TDoubleListImp();
```

Erzeugt eine leere doppelt verkettete Liste.

Template TDoubleListIteratorImp — dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen direkten, doppelt verketteten Listen. Siehe *TMDoubleListIterator*.

Konstruktoren

`TDoubleListIteratorImp(const TDoubleListImp<T> &l)`

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TDoubleListImp*-Objekten.

Template TMSDoubleListImp — dlistimp.h

Implementiert eine sortierte, doppelt verkettete Liste mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik, einen sinnvollen < Operator und einen Standardkonstruktor verfügen.

Protected-Elementfunktionen

Neben den folgenden Elementfunktionen erbt *TMSDoubleListImp* die Elementfunktionen von *TMDoubleListImp*. Siehe *TMDoubleListImp*.

FindDetach `virtual TMDoubleListElement<T> *FindDetach(const T&)`

Siehe *TMDoubleListImp::FindDetach*

FindPred `virtual TMDoubleListElement<T> *FindPred(const T&)`

Siehe *TMDoubleListImp::FindPred*

Template TMSDoubleListIteratorImp — dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen direkten, doppelt verketteten Listen. Siehe *TMDoubleListIterator*.

Konstruktoren

`TMSDoubleListIteratorImp(const TMSDoubleListImp<T,Alloc> &l)`

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TMSDoubleListImp*-Objekten.

Template TSDoubleListImp — dlistimp.h

Implementiert eine sortierte, doppelt verkettete Liste von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik, einen sinn-

vollen < Operator und einen Standardkonstruktor verfügen. Siehe *TMSDoubleListImp*.

Template TSDoubleListIteratorImp — dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen direkten, doppelt verketteten Listen. Siehe *TMDoubleListIterator*.

Konstruktoren

```
TSDoubleListIteratorImp(const TSDoubleListImp<T> &l)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TSDoubleListImp*-Objekten.

Template TMIDoubleListImp — dlistimp.h

Implementiert eine doppelt verkettete Liste mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Die gespeicherten Objekte müssen über einen gültigen == Operator verfügen. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef int ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Public-Elementfunktionen

Add

```
int Add(T *t)
```

Fügt den angegebenen Objektzeiger in die Liste ein.

AddAtHead

```
int AddAdHead(T *t)
```

Fügt den angegebenen Objektzeiger am Listenanfang ein.

AddAtTail `int AddAdTail(T *t)`

Fügt den angegebenen Objektzeiger am Listenende ein.

Detach `int Detach(T *t, int del = 0)`

Entfernt den angegebenen Objektzeiger aus der Liste. Der zweite Parameter entscheidet, ob das Objekt selbst gelöscht wird. Siehe *TShouldDelete*.

DetachAtHead `int DetachAtHead(int del = 0)`

Entfernt den Objektzeiger am Anfang der Liste.

DetachAtTail `int DetachAtTail(int del = 0)`

Entfernt den Objektzeiger am Ende der Liste.

FirstThat `T *FirstThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das erste Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush `void Flush(int = 0)`

Entfernt alle Elemente aus der Liste. Siehe *TMArrayAsVector::Flush*.

ForEach `void ForEach(IterFunc f, void *args)`

ForEach erzeugt einen internen Iterator, mit dem die durch *f* angegebene Funktion auf jedes Listenelement angewendet wird. Siehe *TMArrayAsVector::ForEach*.

GetItemsInContainer `unsigned GetItemsInContainer() const`

Liefert die Anzahl der Listenelemente zurück.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Liste kein Element enthält. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*

PeekHead `T *PeekHead() const`

Liefert den Objektzeiger am Anfang der Liste (Datenelement *Head*) zurück, ohne das Element aus der Liste zu entfernen.

PeekTail `T *PeekTail() const`

Liefert den Objektzeiger am Ende (Datenenelement *Tail*) zurück, ohne das Element aus der Liste zu entfernen.

Protected-Elementfunktionen

FindPred `virtual TIDoubleListElement * *FindPred(void *)`

Siehe *TDoubleListImp::FindPred*

Template TMIDoubleListIteratorImp — dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen indirekten, doppelt verketteten Listen. Für direkte Listen ist *TMDoubleListIterator* vorgesehen.

Konstruktoren

`TMIDoubleListIteratorImp(const TMIDoubleListImp<T,Alloc> &l)`

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TMIDoubleListImp*-Objekten.

Public-Elementfunktionen

Current `T *Current();`

Liefert den Zeiger auf das aktuelle Objekt zurück.

Restart `void Restart();`

Startet die Iteration beginnend mit dem Listenanfang erneut.

Operatoren

operator ++ `T *operator ++(int);`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

`T *operator ++();`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

Template TIDoubleListImp — dlistimp.h

Implementiert eine doppelt verkettete Liste von Zeigern auf Objekte des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Da Zei-

ger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Siehe *TMIDoubleListImp*.

Template TIDoubleListIteratorImp dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen indirekten, doppelt verketteten Listen. Siehe *TMIDoubleListIteratorImp*.

Konstruktoren

```
TIDoubleListIteratorImp(const TIDoubleListImp<T> &l)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TIDoubleListImp*-Objekten.

Template TMISDoubleListImp dlistimp.h

Implementiert eine sortierte doppelt verkettete Liste mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten.

Protected-Elementfunktionen

Neben den hier beschriebenen Elementfunktionen erbt *TMISDoubleListImp* die Elementfunktionen von *TMIDoubleListImp*.

FindDetach

```
virtual TMDoubleListElement<T> *FindDetach(void *)
```

Stellt fest, ob sich ein Element in der Liste befindet, und liefert einen Zeiger auf seine Vorgänger zurück. Wird das Element nicht gefunden, so liefert *FindDetach* den Wert 0 zurück.

Template TMISDoubleListIteratorImp dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen indirekten, sortierten doppelt verketteten Listen. Siehe *TMIDoubleListIteratorImp*.

Konstruktoren

`TMISDoubleListIteratorImp(const TMISDoubleListImp<T,Alloc> &l)`

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TMISDoubleListImp*-Objekten.

Template TISDoubleListImp dlistimp.h

Implementiert eine sortierte, doppelt verkettete Liste von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Siehe *TMIDoubleListImp*.

Template TISDoubleListIteratorImp dlistimp.h

Implementiert ein Iterator-Objekt für doppelt verkettete Listen. Dieser Iterator arbeitet mit allen indirekten, sortierten doppelt verketteten Listen. Siehe *TMIDoubleListIteratorImp*.

Konstruktoren

`TISDoubleListIteratorImp(const TISDoubleListImp<T> &l)`

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TISDoubleListImp*-Objekten.

Template TMHashTableImp hashimp.h

Implementiert eine Hash-Tabelle mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung. *T* muß neben einem Standardkonstruktor einen gültigen == Operator und eine sinnvolle Kopiersemantik besitzen.

Konstruktoren

`TMHashTableImp(unsigned aPrime = DEFAULT_HASH_TABLE_SIZE)`

Erzeugt eine Hash-Tabelle.

Destruktor

`~TMHashTableImp()`

Löscht die Hash-Tabelle durch Aufruf der Elementfunktion *Flush*.

Public-Elementfunktionen

Add

`int Add(const T& t)`

Fügt *t* in die Hash-Tabelle ein.

Detach

`int Detach(const T& t, int del = 0)`

Entfernt das Element *t* aus der Hash-Tabelle. *t* selbst wird gelöscht, falls *del* den Wert 0 enthält. Enthält *del* den Wert 1, so wird es nicht gelöscht.

Find

`T *Find(const T& t) const`

Liefert einen Zeiger auf das Tabellenelement *t*.

Flush

`void Flush(int del = 0)`

Entfernt alle Elemente aus der Hash-Tabelle. Ist der Wert von *del* ungleich null, so wird die Hash-Tabelle selbst auch gelöscht.

ForEach

`void ForEach(void (*f)(T &, void *), void *args)`

ForEach erzeugt einen internen Iterator, mit dem die durch *f* angegebene Funktion auf jedes Tabellenelement angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer

`unsigned GetItemsInContainer() const`

Liefert die aktuelle Anzahl der Elemente in der Hash-Tabelle.

IsEmpty

`int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Hash-Tabelle kein Element enthält. Andernfalls wird 0 zurückgeliefert.

Template TMHashTableIteratorImp — hashimp.h

Implementiert einen Iterator zum Durchlaufen von *TMHashTableImp*-Objekten. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

`TMHashTableIteratorImp(const TMHashTableImp<T,A> &h)`

Erzeugt ein Iterator-Objekt, mit dem *TMHashTableImp*-Objekte durchlaufen werden können.

Destruktor

`~TMHashTableIteratorImp()`

Löscht den Iterator.

Public-Elementfunktionen

Current `const T& Current();`

Liefert das aktuelle Objekt zurück.

Restart `void Restart();`

Startet eine neue Iteration, beginnend mit dem Tabellenanfang.

Operatoren

operator int `operator int() const`

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++ `const T& operator ++(int);`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

`const T& operator ++();`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

Template THashTableImp — hashimp.h

Implementiert eine Hash-Tabelle von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. *T* muß neben einem Standardkonstruktor einen gültigen == Operator und eine sinnvolle Kopiersemantik besitzen. Eine Beschreibung der Klassenelemente finden Sie unter *TMHashTableImp*.

Konstruktoren

`THashTableImp(unsigned aPrime = DEFAULT_HASH_TABLE_SIZE)`

Erzeugt eine Hash-Tabelle, die *TStandardAllocator* zur Speicherverwaltung benutzt.

Template THashTableIteratorImp — hashimp.h

Implementiert einen Iterator zum Durchlaufen von *THashTableImp*-Objekten. Eine Beschreibung der Klassenelemente finden Sie unter *TMHashTableIteratorImp*.

Konstruktoren

`THashTableIteratorImp(const THashTableImp<T> &h)`

Erzeugt ein Iterator-Objekt, mit dem *THashTableImp*-Objekte durchlaufen werden können.

Template TMIHashTableImp — hashimp.h

Implementiert eine Hash-Tabelle mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Die Klasse *A* enthält die vom Benutzer bereitgestellte Speicherverwaltung.

Konstruktoren

`TMIHashTableImp(unsigned aPrime = DEFAULT_HASH_TABLE_SIZE)`

Erzeugt eine indirekte Hash-Tabelle.

Public-Elementfunktionen

Add `int Add(T *t)`

Fügt den Zeiger auf das Element *t* in die Hash-Tabelle ein.

Detach `int Detach(T *t, int del = 0)`

Entfernt den Zeiger auf das Element *t* aus der Hash-Tabelle. *t* selbst wird gelöscht, falls *del* den Wert 0 enthält. Enthält *del* den Wert 1, so wird es nicht gelöscht.

Find `T *Find(T *t) const`

Liefert einen Zeiger auf das Tabellenelement *t*.

Flush `void Flush(int del = 0)`

Entfernt alle Elemente aus der Hash-Tabelle. Ist der Wert von *del* ungleich null, so wird die Hash-Tabelle selbst auch gelöscht.

ForEach `void ForEach(void (*f)(T &, void *), void *args)`

ForEach erzeugt einen internen Iterator, mit dem die durch *f* angegebene Funktion auf jedes Tabellenelement angewendet wird. Über den Parameter *args* können Sie dieser Funktion beliebige Daten übergeben.

GetItemsInContainer `unsigned GetItemsInContainer() const`

Liefert die aktuelle Anzahl der Elemente in der Hash-Tabelle.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Hash-Tabelle kein Element enthält. Andernfalls wird 0 zurückgeliefert.

Template TMIHashTableIteratorImp hashimp.h

Implementiert einen Iterator zum Durchlaufen von *TMIHashTableImp*-Objekten.

Konstruktoren

`TMIHashTableIteratorImp(const TMIHashTableImp<T,A> &h)`

Erzeugt ein Iterator-Objekt, mit dem *TMIHashTableImp*-Objekte durchlaufen werden können.

Public-Elementfunktionen

Current `T *Current();`

Liefert einen Zeiger auf das aktuelle Objekt zurück.

Restart `void Restart();`

Startet eine neue Iteration, beginnend mit dem Tabellenanfang.

Operatoren

operator int

```
operator int() const
```

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++

```
T *operator ++(int);
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert den *vor* der Verschiebung aktuellen Objektzeiger zurück (Post-Inkrement).

```
T *operator ++();
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert den *nach* der Verschiebung aktuellen Objektzeiger zurück (Prä-Inkrement).

Template TIHashTableImp — hashimp.h

Implementiert eine Hash-Tabelle von Zeigern auf Objekte des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Klassenelemente finden Sie unter *TMIHashTableImp*.

Konstruktoren

```
TIHashTableImp(unsigned aPrime = DEFAULT_HASH_TABLE_SIZE)
```

Erzeugt eine indirekte Hash-Tabelle, die *TStandardAllocator* zur Speicherverwaltung benutzt.

Template TIHashTableIteratorImp — hashimp.h

Implementiert einen Iterator zum Durchlaufen von *TIHashTableImp*-Objekten, wobei *TStandardAllocator* zur Speicherverwaltung benutzt wird. Eine Beschreibung der Klassenelemente finden Sie unter *TMIHashTableIteratorImp*.

Konstruktoren

```
TIHashTableIteratorImp(const TIHashTableImp<T> &h)
```

Erzeugt ein Iterator-Objekt, mit dem *TIHashTableImp*-Objekte durchlaufen werden können.

Template TMListElement listimp.h

Diese Klasse definiert die Knoten für die Klassen *TMListImp*, *TIListImp* und alle verwandten Klassen.

Public-Datenelemente

data

```
T data;
```

Datenobjekt, das in Liste enthalten ist.

Next

```
TMListElement<T,Alloc> *Next;
```

Ein Zeiger auf das nächste Element in der Liste.

Konstruktoren

```
TMListElement();
```

Erzeugt ein Listenelement.

```
TMListElement(T& t, TMListElement<T,Alloc> *p);
```

Erzeugt ein Listenelement und fügt es in die Liste hinter dem Objekt ein, auf das *p* zeigt.

Operatoren

operator delete

```
void operator delete(void *);
```

Löscht ein Objekt.

operator new

```
void *operator new(size_t sz);
```

Reserviert einen Speicherblock der Größe *sz* und liefert einen Zeiger auf den Speicherblock zurück.

Template TMListImp listimp.h

Implementiert eine Liste von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen.

Typdefinitionen

CondFunc

`typedef int ( *CondFunc)(const T &, void *);`

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

`typedef int ( *IterFunc)(T &, void *);`

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

`TMListImp()`

Erzeugt eine leere Liste.

Public-Elementfunktionen

Add

`int Add(const T&)`

Fügt das angegebene Objekt in die Liste ein.

Detach

`int Detach(const T& , int = 0)`

Entfernt das angegebene Objekt aus der Liste. Bei erfolgreicher Ausführung wird der Wert 1, im Fehlerfall der Wert 0 zurückgeliefert. Der Integerparameter entscheidet, ob das aus der Liste entfernte Element selbst gelöscht wird. Einzelheiten finden Sie unter *TShouldDelete*.

FirstThat

`T *FirstThat(CondFunc f, void *), void *args) const`

Liefert einen Zeiger auf das erste Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush

`void Flush(int = 0)`

Entfernt alle Elemente aus der Liste, ohne jedoch die Liste selbst zu löschen. Das »Schicksal« der Listenelemente wird vom aktuellen Besitzerstatus und vom Wert des zweiten Parameters bestimmt.

Siehe auch: *TShouldDelete::ownsElements*

ForEach

`void ForEach(IterFunc f, void *args)`

ForEach erzeugt einen internen Iterator, mit dem die durch *f* angegebene Funktion auf jedes Listenelement angewendet wird. Siehe *TMArrayAsVector::ForEach*.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Liste kein Element enthält. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*

PeekHead `const T& PeekHead() const`

Liefert eine Referenz auf das Datenelement *Head* zurück, ohne das Element aus der Liste zu entfernen.

Protected-Datenelemente

Head, Tail `TMListElement<T,Alloc> Head, Tail;`

Die Elemente vor dem ersten und nach dem letzten Listenelement.

Protected-Elementfunktionen

FindDetach `virtual TMListElement<T,Alloc> *FindDetach(const T& t)`

Siehe *TDoubleListImp::FindDetach*

FindPred `virtual TMListElement<T,Alloc> *FindPred(const T&)`

Siehe *TDoubleListImp::FindPred*

Template TMListIteratorImp listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von direkten Listen mit eigener Speicherverwaltung. Für indirekte Listen ist *TMIListIteratorImp* vorgesehen.

Konstruktoren

`TMListIteratorImp(const TMListImp<T,Alloc> &l)`

Erzeugt ein Iterator-Objekt, mit dem *TMListImp*-Objekte durchlaufen werden können.

Public-Elementfunktionen

Current

`const T& Current();`

Liefert das aktuelle Objekt zurück.

Restart

`void Restart();`

Startet eine neue Iteration, beginnend mit dem Listenanfang.

Operatoren

operator int

`operator int() const`

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

operator ++

`const T& operator ++(int);`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

`const T& operator ++();`

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

Template TListImp — listimp.h

Implementiert eine Liste von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen. Eine Beschreibung der Klassenelemente finden Sie unter *TMListImp*.

Template TListIteratorImp — listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von direkten Listen mit eigener Speicherverwaltung. Siehe *TMListIteratorImp*.

Konstruktoren

`TListIteratorImp(const TMListImp<T,TStandardAllocator> &l)`

Erzeugt ein Iterator-Objekt, mit dem *TListImp*-Objekte durchlaufen werden können.

Template TMSListImp listimp.h

Implementiert eine sortierte Liste mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik, einen gültigen < Operator und einen Standardkonstruktor verfügen. Eine Beschreibung der Klassenelemente finden Sie unter *TMListImp*.

Template TMSListIteratorImp listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von sortierten direkten Listen mit eigener Speicherverwaltung. Siehe *TMListIteratorImp*.

Konstruktoren

`TMSListIteratorImp(const TMSListImp<T,Alloc> &l)`

Erzeugt ein Iterator-Objekt, mit dem *TMSListImp*-Objekte durchlaufen werden können.

Template TSListImp listimp.h

Implementiert eine sortierte Liste von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Die Klasse T muß über eine sinnvolle Kopiersemantik, einen gültigen < Operator und einen Standardkonstruktor verfügen. Eine Beschreibung der Klassenelemente finden Sie unter *TMListImp*.

Template TSListIteratorImp listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von sortierten direkten Listen. Siehe *TMListIteratorImp*.

Template TMIListImp listimp.h

Implementiert eine Liste mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef void ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Public-Elementfunktionen

Add

```
int Add(T *t)
```

Fügt den angegebenen Objektzeiger in die Liste ein.

Detach

```
int Detach(T *t, int del = 0)
```

Entfernt den angegebenen Objektzeiger aus der Liste. Der zweite Parameter entscheidet, ob das aus der Liste entfernte Element selbst gelöscht wird. Einzelheiten finden Sie unter *TShouldDelete*.

FirstThat

```
T *FirstThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das erste Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

ForEach

```
void ForEach(IterFunc f, void *args)
```

ForEach erzeugt einen internen Iterator, mit dem die durch *f* angegebene Funktion auf jedes Listenelement angewendet wird. Siehe *TMArrayAsVector::ForEach*.

LastThat

```
T *LastThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das letzte Listenobjekt, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*

PeekHead

```
T *PeekHead() const
```

Liefert einen Zeiger auf das erste Listenelement (*Head*) zurück, ohne das Element aus der Liste zu entfernen.

Protected-Elementfunktionen

FindPred

```
virtual TMListElement<VoidPointer,Alloc> *FindPred(VoidPointer)
```

Siehe *TDoubleListImp::FindPred*.

Template TMIListIteratorImp listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von indirekten Listen mit eigener Speicherverwaltung. Für direkte Listen ist *TMListIteratorImp* vorgesehen.

Konstruktoren

```
TMIListIteratorImp(const TMIListImp<VoidPointer,Alloc> &l)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIListImp*-Objekte durchlaufen werden können.

Public-Elementfunktionen

Current

```
T *Current();
```

Liefert den aktuellen Objektzeiger zurück.

Restart

```
void Restart();
```

Startet eine neue Iteration, beginnend mit dem Listenanfang.

Operatoren

operator ++

```
T *operator ++(int);
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert den *vor* der Verschiebung aktuellen Objektzeiger zurück (Post-Inkrement).

```
T *operator ++();
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert den *nach* der Verschiebung aktuellen Objektzeiger zurück (Prä-Inkrement).

Template TIListImp listimp.h

Implementiert eine Liste von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Siehe *TMIListImp*.

Template TIListIteratorImp listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von indirekten Listen. Siehe *TMIListIteratorImp*.

Konstruktoren

```
TIListIteratorImp(const TIListImp<T> &l)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIListImp*-Objekte durchlaufen werden können.

Template TMISListImp — listimp.h

Implementiert eine sortierte Liste mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten.

Public-Elementfunktionen

FindDetach

```
virtual TMListElement<TVoidPointer,Alloc> *FindDetach(TVoidPointer);
```

Siehe *TDoubleListElement::FindDetach.*

FindPred

```
virtual TMListElement<TVoidPointer,Alloc> *FindPred(TVoidPointer);
```

Siehe *TDoubleListElement::FindDetach.*

Template TMISListIteratorImp — listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von indirekten Listen mit eigener Speicherverwaltung. Für direkte Listen ist *TMIListIteratorImp* vorgesehen.

Konstruktoren

```
TMISListIteratorImp(const TMISListImp<T,Alloc> &l)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TMISListImp*-Objekten.

Template TISListImp — listimp.h

Implementiert eine sortierte Liste von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Siehe *TMISListImp*.

Template TISListIteratorImp listimp.h

Implementiert einen Listen-Iterator zum Durchlaufen von indirekten Listen. Siehe *TMIListIteratorImp*.

Konstruktoren

```
TISListIteratorImp(const TISListImp<T> &l)
```

Erzeugt ein Iterator-Objekt zum Durchlaufen von *TISListImp*-Objekten.

Template TMQueueAsVector queues.h

Implementiert auf Basis eines Vektors eine Queue (Warteschlange) mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik, einen gültigen **<** Operator und einen Standardkonstruktor verfügen.

Konstruktoren

```
TMQueueAsVector(unsigned sz = DEFAULT_QUEUE_SIZE)
```

Erzeugt eine Queue mit eigener Speicherverwaltung der Größe *sz* auf Basis eines Vektors.

Public-Elementfunktionen

FirstThat

```
T *FirstThat(CondFunc, void *args) const
```

Liefert einen Zeiger auf das erste Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush

```
void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)
```

Entfernt alle Elemente aus der Queue, ohne sie jedoch zu löschen. Das »Schicksal« der Stack-Elemente wird vom aktuellen Besitzerstatus und vom Wert des Parameters *dt* bestimmt.

Siehe auch: *TShouldDelete::ownsElements*

ForEach

```
void ForEach(IterFunc f, void *args)
```

Wendet die Funktion *f* auf alle Queue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

Get `T Get();`

Entfernt das Objekt am Ende der Queue. Bei leerer Queue wird 0, ansonsten das aus der Queue entfernte Objekt zurückgeliefert.

GetItemsInContainer `int GetItemsInContainer() const`

Liefert die Anzahl der Elemente in der Queue zurück.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Queue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 1 zurück, falls die Queue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*.

Put `void Put(T t);`

Fügt das angegebene Objekt in die Queue ein.

Template TMQueueAsVectorIterator queues.h

Implementiert ein Iterator-Objekt zum Durchlaufen von Queues auf Vektorbasis mit eigener Speicherverwaltung. Siehe *TMDequeAsVectorIterator*.

Konstruktoren

```
TMQueueAsVectorIterator(const TMDequeAsVector<T,Alloc> &q)
```

Erzeugt ein Iterator-Objekt, mit dem *TMQueueAsVector*-Objekte durchlaufen werden können.

Template TQueueAsVector queues.h

Siehe *TMQueueAsVector* für eine Beschreibung der Klassenelemente.

Konstruktoren

```
TQueueAsVector(unsigned sz = DEFAULT_QUEUE_SIZE)
```

Erzeugt eine Queue der Größe *sz* auf Vektorbasis.

Template TQueueAsVectorIterator — queues.h

Implementiert ein Iterator-Objekt für Queues auf Vektorbasis. Siehe *TMDequeAsVectorIterator*.

Konstruktoren

`TQueueAsVectorIterator(const TQueueAsVector<T> &q)`

Erzeugt ein Iterator-Objekt, mit dem *TQueueAsVector*-Objekte durchlaufen werden können.

Template TMIQueueAsVector — queues.h

Implementiert auf Basis eines Vektors eine Queue (Warteschlange) mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*.

Konstruktoren

`TMIQueueAsVector(unsigned sz = DEFAULT_QUEUE_SIZE)`

Erzeugt eine indirekte Queue mit eigener Speicherverwaltung der Größe *sz*.

Public-Elementfunktionen

FirstThat

`T *FirstThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das erste Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush

`void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)`

Siehe *TMQueueAsVector::Flush*.

ForEach

`void ForEach(IterFunc f, void *args)`

Wendet die Funktion *f* auf alle Queue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

Get

`T *Get();`

Entfernt den Objekt-Zeiger am Ende der Queue und liefert ihn zurück. Bei leerer Queue wird 0, ansonsten das aus der Queue entfernte Objekt zurückgeliefert.

GetItemsInContainer `int GetItemsInContainer() const`

Liefert die Anzahl der Elemente in der Queue zurück.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Queue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 1 zurück, falls die Queue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*.

Put `void Put(T *t);`

Fügt einen Objekt-Zeiger am Ende der Queue ein.

Template TMIQueueAsVectorIterator queues.h

Implementiert ein Iterator-Objekt zum Durchlaufen von auf Vektoren basierenden, indirekten Queues mit eigener Speicherverwaltung.

Konstruktoren

`TMIQueueAsVectorIterator(const TMIDequeAsVector<T,Alloc> &q)`

Erzeugt ein Iterator-Objekt, mit dem *TMIQueueAsVector*-Objekte durchlaufen werden können.

Template TIQueueAsVector queues.h

Implementiert auf Basis eines Vektors eine Queue (Warteschlange) von Zeigern auf Objekte des Typs *T*.

Konstruktoren

`TIQueueAsVector(unsigned sz = DEFAULT_QUEUE_SIZE)`

Erzeugt eine indirekte Queue der Größe *sz*.

Template TIQueueAsVectorIterator queues.h

Implementiert ein Iterator-Objekt zum Durchlaufen von auf Vektoren basierenden, indirekten Queues. Siehe *TMDequeAsVectorIterator*.

Konstruktoren

`TIQueueAsVectorIterator(const TIQueueAsVector<T> &q)`

Erzeugt ein Iterator-Objekt, mit dem *TIQueueAsVector*-Objekte durchlaufen werden können.

Template TMQueueAsDoubleList queues.h

Implementiert auf Basis einer doppelt verketteten Liste eine Queue (Warteschlange) mit eigener Speicherverwaltung von Objekten des Typs *T*. Eine Beschreibung der Datenelemente finden Sie unter *TMDequeAsDoubleList*.

Public-Elementfunktionen

FirstThat `T *FirstThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das erste Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush `void Flush(int del)`

Entfernt alle Elemente aus der Queue. Siehe *TMQueueAsVector::Flush*.

ForEach `void ForEach(IterFunc f, void *args)`

Wendet die Funktion *f* auf alle Queue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

Get `T Get()`

Entfernt das Objekt am Ende der Queue. Bei leerer Queue wird 0, ansonsten das aus der Queue entfernte Objekt zurückgeliefert.

GetItemsInContainer `int GetItemsInContainer() const`

Liefert die Anzahl der Elemente in der Queue zurück.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls die Queue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 1 zurück, falls die Queue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*.

Put `void Put(T t);`

Fügt das angegebene Objekt an das Ende der Queue ein.

Template TMQueueAsDoubleListIterator queues.h

Implementiert ein Iterator-Objekt zum Durchlaufen von auf Listen basierenden Queues mit eigener Speicherverwaltung.

Konstruktoren

`TMQueueAsDoubleListIterator(const TMQueueAsDoubleList<T,Alloc> &q)`

Erzeugt ein Iterator-Objekt, mit dem *TMQueueAsDoubleList*-Objekte durchlaufen werden können.

Template TQueueAsDoubleList queues.h

Implementiert auf Basis einer doppelt verketten Liste eine Queue (Warteschlange) von Objekten des Typs *T*. Eine Beschreibung der Klassenelemente finden Sie unter *TMQueueAsDoubleList*.

Template TQueueAsDoubleListIterator queues.h

Implementiert ein Iterator-Objekt zum Durchlaufen von Queues, die auf doppelt verketteten Listen basieren. Siehe *TMDequeAsDoubleListIterator*.

Konstruktoren

`TQueueAsDoubleListIterator(const TQueueAsDoubleList<T> & q)`

Erzeugt ein Iterator-Objekt, mit dem *TQueueAsDoubleList*-Objekte durchlaufen werden können.

Implementiert auf Basis einer doppelt verketten Liste eine indirekte Queue (Warteschlange) mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*.

Public-Elementfunktionen

FirstThat

```
T *FirstThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das erste Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush

```
void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)
```

Entfernt alle Elemente aus der Queue. Siehe *TMQueueAsVector::Flush*.

ForEach

```
void ForEach(IterFunc f, void *args)
```

Wendet die Funktion *f* auf alle Queue-Elemente an. Siehe *TMArrayAsVector::ForEach*.

Get

```
T *Get();
```

Entfernt den Objekt-Zeiger am Ende der Queue und liefert ihn zurück. Bei leerer Queue wird 0 zurückgeliefert.

GetItemsInContainer

```
int GetItemsInContainer() const
```

Liefert die Anzahl der Elemente in der Queue zurück.

IsEmpty

```
int IsEmpty() const
```

Liefert den Wert 1 zurück, falls die Queue kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull

```
int IsFull() const
```

Liefert den Wert 1 zurück, falls die Queue voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat

```
T *LastThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das letzte Queue-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*.

Put

```
void Put(T *t);
```

Fügt das angegebene Objekt am Ende in die Queue ein.

Template TMIQueueAsDoubleListIterator queues.h

Implementiert ein Iterator-Objekt zum Durchlaufen von indirekten Queues, die auf doppelt verketteten Listen basieren. Siehe TMIDequeAsDoubleListIterator.

Konstruktoren

```
TMIQueueAsDoubleListIterator(const TMIQueueAsDoubleList<T, Alloc> & q)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIQueueAsDoubleList*-Objekte durchlaufen werden können.

Template TIQueueAsDoubleList queues.h

Implementiert auf Basis einer doppelt verketten Liste eine indirekte Queue (Warteschlange) von Zeigern auf Objekte des Typs *T*. Siehe *TMIQueueAsDoubleList*.

Template TIQueueAsDoubleListIterator queues.h

Implementiert ein Iterator-Objekt zum Durchlaufen von indirekten Queues, die auf doppelt verketteten Listen basieren. Siehe *TMIDequeAsDoubleListIterator*.

Konstruktoren

```
TIQueueAsDoubleListIterator(const TIQueueAsDoubleList<T> & q)
```

Erzeugt ein Iterator-Objekt, mit dem *TIQueueAsDoubleList*-Objekte durchlaufen werden können.

Template TQueue queues.h

Kurzschreibweise für *TQueueAsVector*.

Template TQueueIterator queues.h

Kurzschreibweise für *TQueueAsVectorIterator*.

Template TMSetAsVector sets.h

Implementiert auf Vektorbasis einen Set mit eigener Speicherverwaltung von Objekten des Typs *T*. Ein Set kann – im Gegensatz zum Bag – keine doppelten Elemente enthalten.

Konstruktoren

```
TMSetAsVector (unsigned sz=DEFAULT_SET_SIZE)
```

Erzeugt einen leeren Set. Der Parameter *sz* enthält die Anzahl der Elemente, die der Set aufnehmen kann.

Public-Elementfunktionen

Neben den folgenden erbt *TMSetAsVector* weitere Elementfunktionen von *TMBagAsVector*. Siehe *TMBagAsVector*.

Add

```
int Add(const T&);
```

Fügt das angegebene Objekt dem Set hinzu.

Template TMSetAsVectorIterator sets.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TMSetAsVector*-Objekten. Eine Beschreibung der Operatoren und Elementfunktionen finden Sie unter *TMArrayAsVectorIterator*.

Konstruktoren

```
TMSetAsVectorIterator(const TMSetAsVector<T,Alloc> &s)
```

Erzeugt ein Iterator-Objekt, mit dem *TMSetAsVector*-Objekte durchlaufen werden können.

Template TSetAsVector sets.h

Implementiert auf Vektorbasis einen Set von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Operatoren und Elementfunktionen finden Sie unter *TMBagAsVector*.

Konstruktoren

```
TSetAsVector (unsigned sz=DEFAULT_SET_SIZE) :
```

Erzeugt einen leeren Set. Der Parameter *sz* enthält die Anzahl der Elemente, die der Set aufnehmen kann.

Template TSetAsVectorIterator sets.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TSetAsVector*-Objekten. Eine Beschreibung der Operatoren und Elementfunktionen finden Sie unter *TMArrayAsVectorIterator*.

Konstruktoren

```
TSetAsVectorIterator(const TSetAsVector<T> &s)
```

Erzeugt ein Iterator-Objekt, mit dem *TSetAsVector*-Objekte durchlaufen werden können.

Template TMISetAsVector sets.h

Implementiert auf Vektorbasis einen Set mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Eine Beschreibung der Klassenelemente finden Sie unter *TMIBagAsVector*.

Konstruktoren

```
TMISetAsVector(unsigned sz = DEFAULT_SET_SIZE)
```

Erzeugt einen leeren, indirekten Set mit eigener Speicherverwaltung. Der Parameter *sz* enthält die Anzahl der Elemente, die der Set aufnehmen kann.

Public-Elementfunktionen

Neben den folgenden erbt *TMISetAsVector* weitere Elementfunktionen von *TMIBagAsVector*. Siehe *TMIBagAsVector*.

Add

```
int Add(T *);
```

Fügt den angegebenen Objekt-Zeiger dem Set hinzu.

Template TMISetAsVectorIterator — sets.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TMISetAsVector*-Objekten. Eine Beschreibung der Operatoren und Elementfunktionen finden Sie unter *TMIArrayAsVectorIterator*.

Konstruktoren

```
TMISetAsVectorIterator(const TMISetAsVector<T,Alloc> &s)
```

Erzeugt ein Iterator-Objekt, mit dem *TMISetAsVector*-Objekte durchlaufen werden können.

Template TISetAsVector — sets.h

Implementiert auf Vektorbasis einen Set von Zeigern auf Objekte des Typs *T*. Eine Beschreibung der Klassenelemente finden Sie unter *TMIBagAsVector*.

Konstruktoren

```
TISetAsVector(unsigned sz = DEFAULT_SET_SIZE)
```

Erzeugt einen leeren, indirekten Set. Der Parameter *sz* enthält die Anzahl der Elemente, die der Set aufnehmen kann.

Template TISetAsVectorIterator — sets.h

Implementiert ein Iterator-Objekt zum Durchlaufen von *TISetAsVector*-Objekten. Eine Beschreibung der Operatoren und Elementfunktionen finden Sie unter *TMIArrayAsVectorIterator*.

Konstruktoren

```
TISetAsVectorIterator(const TISetAsVector<T> &s)
```

Erzeugt ein Iterator-Objekt, mit dem *TISetAsVector*-Objekte durchlaufen werden können.

Template TSet queues.h

Kurzschreibweise für *TSetAsVector*.

Template TSetIterator queues.h

Kurzschreibweise für *TSetAsVectorIterator*.

Template TMStackAsVector stacks.h

Implementiert auf Vektorbasis einen Stack mit eigener Speicherverwaltung von Objekten des Typs *T*.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef void ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

```
TMStackAsVector(unsigned max = DEFAULT_STACK_SIZE)
```

Erzeugt einen Stack mit eigener Speicherverwaltung, der auf einem Vektor basiert. Der Parameter *max* bezeichnet die maximale Stack-Größe.

Public-Elementfunktionen

FirstThat

`T *FirstThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das erste Stack-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat*

Flush

`void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)`

Entfernt alle Elemente aus der Stack, ohne sie jedoch zu löschen. Das »Schicksal« der Stack-Elemente wird vom aktuellen Besitzerstatus und vom Wert des Parameters *dt* bestimmt.

Siehe auch: *TShouldDelete::ownsElements*

ForEach

`void ForEach(IterFunc f, void *args)`

Wendet die Funktion *f* auf alle Stack-Elemente an. Siehe *TMArrayAsVector::ForEach*.

GetItemsInContainer

`int GetItemsInContainer() const`

Liefert die Anzahl der Elemente im Stack zurück.

IsEmpty

`int IsEmpty() const`

Liefert den Wert 1 zurück, falls der Stack kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull

`int IsFull() const`

Liefert den Wert 1 zurück, falls der Stack voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat

`T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Objekt des Stack zurück, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*

Pop

`T Pop()`

Entfernt das Objekt vom oberen Ende des Stapels und liefert es zurück. Das »Schicksal« des entfernten Objekts wird durch seinen Besitzerstatus bestimmt. Siehe *TShouldDelete*.

Push

`void Push(const T& t)`

Legt das angegebene Objekt auf den Stack ab.

Top

`T Top() const`

Liefert das oberste Stack-Element zurück, entfernt es dabei jedoch nicht vom Stack.

Template TMStackAsVectorIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines Stack mit eigener Speicherverwaltung, der auf einem Vektor basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMVectorIteratorImp*.

Konstruktoren

```
TMStackAsVectorIterator(const TMStackAsVector<T,Alloc> & s);
```

Erzeugt ein Iterator-Objekt, mit dem *TMStackAsVector*-Objekte durchlaufen werden können.

Template TStackAsVector stacks.h

Implementiert auf Vektorbasis einen Stack von Objekten des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt.

Konstruktoren

```
TStackAsVector(unsigned max = DEFAULT_STACK_SIZE)
```

Erzeugt einen Stack, der auf einem Vektor basiert. Der Parameter *max* bezeichnet die maximale Stack-Größe.

Template TStackAsVectorIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines Stack, der auf einem Vektor basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMVectorIteratorImp*.

Konstruktoren

```
TStackAsVectorIterator(const TStackAsVector<T> & s);
```

Erzeugt ein Iterator-Objekt, mit dem *TStackAsVector*-Objekte durchlaufen werden können.

Implementiert auf Vektorbasis einen indirekten Stack mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*.

Typdefinitionen

CondFunc

```
typedef void ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef void ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

```
TMIStackAsVector(unsigned max = DEFAULT_STACK_SIZE)
```

Erzeugt einen indirekten Stack mit eigener Speicherverwaltung, der auf einem Vektor basiert. Der Parameter *max* bezeichnet die maximale Stack-Größe.

Public-Elementfunktionen

FirstThat

```
T *FirstThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das erste Stack-Objekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat.*

Flush

```
void Flush(TShouldDelete::DeleteType dt = TShouldDelete::DefDelete)
```

Entfernt alle Elemente aus dem Stack, ohne den Stack jedoch zu löschen. Das »Schicksal« der entfernten Stack-Elemente wird vom aktuellen Besitzerstatus und vom Wert des Parameters *dt* bestimmt.

Siehe auch: *TShouldDelete::ownsElements.*

ForEach

```
void ForEach(IterFunc f, void *args)
```

Wendet die Funktion *f* auf alle Stack-Elemente an. Siehe *TMArrayAsVector::ForEach*.

GetItemsInContainer

```
int GetItemsInContainer() const
```

Liefert die Anzahl der Elemente im Stack zurück.

IsEmpty `int IsEmpty() const`

Liefert den Wert 1 zurück, falls der Stack kein Element enthält. Andernfalls wird 0 zurückgeliefert.

IsFull `int IsFull() const`

Liefert den Wert 1 zurück, falls der Stack voll ist. Andernfalls wird 0 zurückgeliefert.

LastThat `T *LastThat(CondFunc f, void *args) const`

Liefert einen Zeiger auf das letzte Objekt des Stack zurück, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

Siehe auch: *FirstThat*, *ForEach*

Pop `T *Pop()`

Entfernt den Objektzeiger vom oberen Ende des Stapels und liefert ihn zurück. Das »Schicksal« des entfernten Objekts wird durch seinen Besitzerstatus bestimmt. Siehe *TShouldDelete*.

Push `void Push(T *t)`

Legt den angegebene Objektzeiger auf den Stack ab.

Top `T *Top() const`

Liefert das oberste Stack-Element zurück, entfernt es dabei jedoch nicht vom Stack.

Template TMIStackAsVectorIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines indirekten Stack mit eigener Speicherverwaltung, der auf einem Vektor basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMVectorIteratorImp*.

Konstruktoren

```
TMIStackAsVectorIterator(const TMIStackAsVector<T,Alloc> &s);
```

Erzeugt ein Iterator-Objekt, mit dem *TMIStackAsVector*-Objekte durchlaufen werden können.

Template TIStackAsVector stacks.h

Implementiert auf Vektorbasis einen indirekten Stack von Zeigern auf Objekte des Typs *T*. Eine Beschreibung der Klassenelemente finden Sie unter *TMIStackAsVector*.

Konstruktoren

```
TIStackAsVector(unsigned max = DEFAULT_STACK_SIZE) :
TMIStackAsVector<T,TStandardAllocator>(max)
```

Erzeugt einen indirekten Stack, der auf einem Vektor basiert. Der Parameter *max* bezeichnet die maximale Stack-Größe.

Template TIStackAsVectorIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines indirekten Stack, der auf einem Vektor basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMIVectorIteratorImp*.

Konstruktoren

```
TMIStackAsVectorIterator(const TMIStackAsVector<T,Alloc> &s);
```

Erzeugt ein Iterator-Objekt, mit dem *TIStackAsVector*-Objekte durchlaufen werden können.

Template TMStackAsList stacks.h

Implementiert einen Stack mit eigener Speicherverwaltung von Objekten des Typs *T*, der auf einer Liste basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMStackAsVector*.

Template TMStackAsListIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines Stack mit eigener Speicherverwaltung, der auf einer Liste basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMListIteratorImp*.

Konstruktoren

```
TMStackAsListIterator(const TMStackAsList<T,Alloc> &s) :
TMListIteratorImp<T,Alloc>(s.Data)
```

Erzeugt ein Iterator-Objekt, mit dem *TMStackAsList*-Objekte durchlaufen werden können.

Template TStackAsList stacks.h

Implementiert einen Stack von Objekten des Typs *T*, der auf einer Liste basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMStackAsVector*.

Template TStackAsListIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines Stack, der auf einer Liste basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMVectorIteratorImp*.

Konstruktoren

```
TStackAsListIterator(const TStackAsList<T> &s) :
TMStackAsListIterator<T,TStandardAllocator>(s)
```

Erzeugt ein Iterator-Objekt, mit dem *TStackAsList*-Objekte durchlaufen werden können.

Template TMIStackAsList stacks.h

Implementiert auf Basis einer Liste einen indirekten Stack mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Eine Beschreibung der Klassenelemente finden Sie unter *TMIStackAsVector*.

Template TMIStackAsListIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines indirekten Stack, der auf einer Liste basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMIListIteratorImp*.

Konstruktoren

```
TMIStackAsListIterator(const TMIStackAsList<T,Alloc> &s)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIStackAsList*-Objekte durchlaufen werden können.

Template TIStackAsList stacks.h

Implementiert auf Basis einer Liste einen indirekten Stack von Zeigern auf Objekte des Typs *T*. *TStandardAllocator* wird zur Speicherverwaltung benutzt. Eine Beschreibung der Klassenelemente finden Sie unter *TMIStackAsVector*.

Template TIStackAsListIterator stacks.h

Implementiert ein Iterator-Objekt zum Durchlaufen eines indirekten Stack, der auf einer Liste basiert. Eine Beschreibung der Klassenelemente finden Sie unter *TMIVectorIteratorImp*.

Konstruktoren

`TIStackAsListIterator(const TIStackAsList<T> &s)`

Erzeugt ein Iterator-Objekt, mit dem *TIStackAsList*-Objekte durchlaufen werden können.

Template TStack queues.h

Kurzschreibweise für *TStackAsVector*.

Template TStackIterator queues.h

Kurzschreibweise für *TStackAsVectorIterator*.

Template TMVectorImp vectimp.h

Implementiert einen Vektor mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen.

Typdefinitionen

CondFunc

```
typedef int ( *CondFunc)(const T &, void *);
```

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc

```
typedef void ( *IterFunc)(T &, void *);
```

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

```
TMVectorImp();
```

Erzeugt einen Vektor ohne Einträge.

```
TMVectorImp(unsigned sz, unsigned = 0);
```

Erzeugt einen Vektor für *sz* Objekte, die standardmäßig mit 0 initialisiert werden.

```
TMVectorImp(const TMVectorImp<T,Alloc> &);
```

Erzeugt eine Kopie eines Vektors.

Public-Elementfunktionen

FirstThat

```
T *FirstThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das erste Vektorobjekt, das einer gegebenen Bedingung genügt. Siehe *TMArrayAsVector::FirstThat*.

```
T *FirstThat(CondFunc f, void *args, unsigned, unsigned) const
```

Diese Version erlaubt zusätzlich die Angabe eines Suchbereichs. Siehe *TMArrayAsVector::FirstThat*.

Siehe auch: *LastThat.*

Flush

```
void Flush(unsigned = 0, unsigned = UINT_MAX, unsigned = 0);
```

Entfernt alle Elemente aus dem Vektor, ohne den Vektor zu löschen. Das »Schicksal« der Vektorelemente wird vom aktuellen Besitzerstatus und vom Wert des ersten Arguments bestimmt. Über die letzten beiden Parametern kann das Entfernen der Vektorelemente auf einen bestimmten Bereich beschränkt werden.

Siehe auch: *TShouldDelete::ownsElements.*

ForEach

```
void ForEach(IterFunc f, void *args)
```

Wendet die Funktion *f* auf alle Vektorelemente an. Siehe *TMArrayAsVector::ForEach*.

```
void ForEach(IterFunc, void *args, unsigned, unsigned)
```

Diese Version erlaubt zusätzlich die Angabe eines Bereichs.

GetDelta

```
virtual unsigned GetDelta() const;
```

Liefert den Größenzuwachswert *delta* des Vektors zurück.

LastThat

```
T *LastThat(CondFunc f, void *args) const
```

Liefert einen Zeiger auf das letzte Objekt des Vektors zurück, das eine gegebene Bedingung erfüllt. Siehe *TMArrayAsVector::LastThat*.

```
T *LastThat(CondFunc, void *args, unsigned, unsigned) const
```

Diese Version erlaubt zusätzlich die Angabe eines Bereichs.

Siehe auch: *FirstThat*, *ForEach*

Limit

```
unsigned Limit() const;
```

Liefert die maximale Anzahl der Elemente zurück, die der Vektor aufnehmen kann.

Resize

```
void Resize(unsigned sz, unsigned offset = 0);
```

Erzeugt einen neuen Vektor der Größe *sz*. Der bereits bestehende Vektor wird in den neuen Vektor kopiert und dann gelöscht. In einem Vektor von Zeigern werden die Einträge auf NULL gesetzt. In einem Vektor vom Objekten wird für jedes nicht benutzte Vektorelement der Standardkonstruktor aufgerufen. Der Parameter *offset* bezeichnet die Position im neuen Vektor, die das erste Element des alten Vektors aufnehmen soll. Dies wird benötigt, wenn der Vektor nach unten erweitert werden soll.

Top

```
virtual unsigned Top() const;
```

Liefert den Index des obersten Elements zurück. Bei einfachen Vektoren liefert *Top* das Datenelement *lim* zurück, bei abgezählten oder sortierten Vektoren wird die aktuelle Einfügeposition zurückgeliefert.

Operatoren

operator []

```
T & operator [] (unsigned index) const
```

Liefert eine Objektreferenz zurück.

operator =

```
const TMVectorImp<T,Alloc> & operator = (const TMVectorImp<T,Alloc> &)
```

Implementiert den Zuweisungoperator für Vektoren.

Protected-Datenelemente

Lim

```
unsigned Lim;
```

Lim enthält die obere Grenze der gültigen Indexwerte des Vektors.

Protected-Elementfunktionen

Zero

```
virtual void Zero(unsigned, unsigned)
```

Funktion zum Nullsetzen bestimmter Bereiche innerhalb des Vektors.

Template TMVectorIteratorImp — vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von direkten *T*-Objekt-Vektoren mit eigener Speicherverwaltung. Für indirekte Vektoren ist *TMIVectorIteratorImp* vorgesehen.

Konstruktoren

```
TMVectorIteratorImp(const TMVectorImp<T,Alloc> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TMVectorImp*-Objekte durchlaufen werden können.

```
TMVectorIteratorImp(const TMVectorImp<T,Alloc> &v, unsigned start,
                    unsigned stop)
```

Erzeugt ein Iterator-Objekt, mit dem *TMVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Public-Elementfunktionen

Current

```
const T& Current();
```

Liefert das aktuelle Objekt zurück.

Restart

```
void Restart();
```

Startet eine neue Iteration über den gesamten Vektor.

```
void Restart(unsigned start, unsigned stop);
```

Startet eine neue Iteration über den angegebenen Vektorbereich.

Operatoren

operator ++

```
const T& operator ++(int);
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *vor* der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

```
const T& operator ++();
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das *nach* der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

operator int

```
operator int() const
```

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

Template TVectorImp — vectimp.h

Implementiert einen Vektor von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen. Eine Beschreibung der Klassenelemente finden Sie unter *TMVectorImp*.

Konstruktoren

```
TVectorImp();
```

Erzeugt einen Vektor ohne Einträge.

```
TVectorImp(unsigned sz, unsigned = 0);
```

Erzeugt einen Vektor für *sz* Objekte, die standardmäßig mit 0 initialisiert werden.

```
TVectorImp(const TVectorImp<T> &v);
```

Erzeugt eine Kopie eines Vektors.

Template TVectorIteratorImp — vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von direkten *T*-Objekt-Vektoren. Siehe *TMVectorIteratorImp*.

Konstruktoren

```
TVectorIteratorImp(const TVectorImp<T> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TVectorImp*-Objekte durchlaufen werden können.

```
TVectorIteratorImp(const TVectorImp<T> &v, unsigned start, unsigned stop)
```

Erzeugt ein Iterator-Objekt, mit dem *TVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TMCVectorImp — vectimp.h

Implementiert einen abgezählten Vektor mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen.

Konstruktoren

```
TMCVectorImp();
```

Erzeugt einen Vektor ohne Einträge.

```
TMCVectorImp(unsigned sz, unsigned = 0);
```

Erzeugt einen Vektor für *sz* Objekte, die standardmäßig mit 0 initialisiert werden.

Public-Elementfunktionen

Neben den folgenden Elementfunktionen erbt *TMCVectorImp* die Elementfunktionen von *TMVectorImp*. Siehe *TMVectorImp*.

Add

```
int Add(const T&)
```

Fügt ein Objekt dem Vektor hinzu und inkrementiert *Count_*.

AddAt

```
int AddAt(const T&, unsigned)
```

Fügt ein Objekt an einer bestimmten Position in den Vektor ein und inkrementiert *Count_*.

Count

```
unsigned Count() const;
```

Liefert *Count_* zurück.

Detach

```
int Detach(unsigned, int = 0)
int Detach(const T&, int = 0)
```

Entfernt ein Objekt. Die erste Version entfernt das Objekt an der angegebenen Indexposition, die zweite entfernt das übergebene Objekt aus dem Vektor. Der zweite Parameter legt fest, ob das entfernte Objekt gelöscht werden soll. Siehe *TMArrayAsVector::Detach.*

Find

```
virtual unsigned Find (const T&) const;
```

Sucht das angegebene Objekt und liefert seinen Indexwert zurück, sofern es gefunden wird. Ansonsten wird INT_MAX zurückgeliefert.

GetDelta

```
virtual unsigned GetDelta() const
```

Liefert das Datenelement *Delta* zurück.

Protected-Datenelemente

Neben den folgenden erbt *TMCVectorImp* die Datenelemente von *TMVectorImp*. Siehe *TMVectorImp*.

Count_

```
unsigned Count_;
```

Enthält die aktuelle Anzahl von Objekten im Vektor.

Delta

```
unsigned Delta;
```

Bestimmt den Zuwachswert, der bei einer Vergrößerung des Vektors verwendet wird.

Protected-Elementfunktionen

Top

```
virtual unsigned Top() const
```

Liefert *Count_* zurück.

Template TMCVectorIteratorImp — vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von direkten abgezählten Vektoren mit eigener Speicherverwaltung von Objekten des Typs *T*. Siehe *TMVectorIteratorImp*.

Konstruktoren

```
TMCVectorIteratorImp(const TMCVectorImp<T,Alloc> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TMCVectorImp*-Objekte durchlaufen werden können.

```
TMCVectorIteratorImp(const TMCVectorImp<T,Alloc> &v, unsigned start,
unsigned stop)
```

Erzeugt ein Iterator-Objekt, mit dem *TMCVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TCVectorImp vectimp.h

Implementiert einen abgezählten Vektor von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik und einen Standardkonstruktor verfügen. Siehe *TMCVectorImp*.

Konstruktoren

```
TMCVectorImp();
```

Erzeugt einen Vektor ohne Einträge.

```
TMCVectorImp(unsigned sz, unsigned = 0);
```

Erzeugt einen Vektor für *sz* Objekte, die standardmäßig mit 0 initialisiert werden.

Template TCVectorIteratorImp vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von direkten abgezählten Vektoren von Objekten des Typs *T*. Siehe *TMCVectorIteratorImp*.

Konstruktoren

```
TCVectorIteratorImp(const TCVectorImp<T> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TCVectorImp*-Objekte durchlaufen werden können.

```
TCVectorIteratorImp(const TCVectorImp<T> &v, unsigned start, unsigned stop)
```

Erzeugt ein Iterator-Objekt, mit dem *TCVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TMSVectorImp vectimp.h

Implementiert einen sortierten Vektor mit eigener Speicherverwaltung von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik, einen sinnvollen < Operator und einen Standardkonstruktor verfügen. Eine Beschreibung der Datenelemente finden Sie unter *TMCVectorImp*.

Konstruktoren

```
TMSVectorImp();
```

Erzeugt einen Vektor ohne Einträge.

```
TMSVectorImp(unsigned sz, unsigned = 0);
```

Erzeugt einen Vektor für *sz* Objekte, die standardmäßig mit 0 initialisiert werden.

Template TMSVectorIteratorImp vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von direkten sortierten Vektoren mit eigener Speicherverwaltung von Objekten des Typs T. Siehe *TMVectorIteratorImp*.

Konstruktoren

```
TMSVectorIteratorImp(const TMSVectorImp<T,Alloc> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TMSVectorImp-Objekte* durchlaufen werden können.

```
TMSVectorIteratorImp(const TMSVectorImp<T,Alloc> &v, unsigned start,
unsigned stop)
```

Erzeugt ein Iterator-Objekt, mit dem *TMSVectorImp-Objekte* durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TSVectorImp vectimp.h

Implementiert einen sortierten Vektor von Objekten des Typs *T*. Die Klasse *T* muß über eine sinnvolle Kopiersemantik, einen sinnvollen < Operator und einen Standardkonstruktor verfügen. Eine Beschreibung der Klassenelemente finden Sie unter *TMCVectorImp*.

Konstruktoren

`TSVectorImp();`

Erzeugt einen Vektor ohne Einträge.

`TSVectorImp(unsigned sz, unsigned = 0);`

Erzeugt einen Vektor für *sz* Objekte, die standardmäßig mit 0 initialisiert werden.

Template TSVectorIteratorImp vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von direkten sortierten Vektoren von Objekten des Typs *T*. Siehe *TMVectorIteratorImp*.

Konstruktoren

`TSVectorIteratorImp(const TSVectorImp<T> &v)`

Erzeugt ein Iterator-Objekt, mit dem *TSVectorImp*-Objekte durchlaufen werden können.

`TSVectorIteratorImp(const TSVectorImp<T> &v, unsigned start, unsigned stop)`

Erzeugt ein Iterator-Objekt, mit dem *TSVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TMIVectorImp vectimp.h

Implementiert einen Vektor mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs T. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten.

Typdefinitionen

CondFunc `typedef int (*CondFunc)(const T &, void *);`

Funktionstyp, der für die Elementfunktionen *FirstThat* und *LastThat* als Parameter verwendet wird.

IterFunc `typedef void (*IterFunc)(T &, void *);`

Funktionstyp, der für die Elementfunktion *ForEach* als Parameter verwendet wird.

Konstruktoren

```
TMIVectorImp(unsigned sz);
```

Erzeugt einen Vektor mit eigener Speicherverwaltung für *sz* Objektzeiger.

Public-Funktionstypen

FirstThat

```
T *FirstThat(CondFunc f, void *args) const
T *FirstThat(CondFunc f, void *args, unsigned, unsigned) const
```

Siehe *TMVectorImp::FirstThat*.

Flush

```
void Flush(unsigned = 0, unsigned = UINT_MAX, unsigned = 0);
```

Siehe *TMVectorImp::Flush*.

ForEach

```
void ForEach(IterFunc f, void *args)
void ForEach(IterFunc f, void *args, unsigned, unsigned)
```

Siehe *TMVectorImp::ForEach.*

GetDelta

```
virtual unsigned GetDelta() const;
```

Liefert den Größenzuwachswert *Delta* zurück.

LastThat

```
T *LastThat(CondFunc f, void *args) const
T *LastThat(CondFunc f, void *args, unsigned, unsigned) const
```

Siehe *TMVectorImp::LastThat*.

Limit

```
unsigned Limit() const;
```

Liefert die maximale Anzahl der Elemente zurück, die der Vektor aufnehmen kann.

Resize

```
Resize(unsigned sz, unsigned offset = 0);
```

Erzeugt einen neuen Vektor der Größe *sz*. Der bereits bestehende Vektor wird in den neuen Vektor kopiert und dann gelöscht. In einem Vektor von Zeigern werden die Einträge auf NULL gesetzt. In einem Vektor von Objekten wird für jedes nicht benutzte Vektorelement der Standardkonstruktor aufgerufen. Der Parameter *offset* bezeichnet die Position im neuen Vektor, die das erste Element des alten Vektors aufnehmen soll. Dies wird benötigt, wenn der Vektor nach unten erweitert werden soll.

Top

```
virtual unsigned Top() const;
```

Liefert den Index des obersten Elements zurück. Bei einfachen Vektoren liefert *Top* das Datenelement *lim* zurück, bei abgezählten oder sortierten Vektoren wird die aktuelle Einfügeposition zurückgeliefert.

Zero

```
virtual void Zero(unsigned, unsigned)
```

Siehe *TMVectorImp::Zero*.

Operatoren

operator []

```
T & operator [] (unsigned index)
T & operator [] (unsigned index) const
```

Siehe *TMVectorImp*, Operator []

Template TMIVectorIteratorImp vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von indirekten T-Objekt-Vektoren mit eigener Speicherverwaltung.

Konstruktoren

```
TMIVectorIteratorImp(const TMIVectorImp<T,Alloc> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIVectorImp*-Objekte durchlaufen werden können.

```
TMIVectorIteratorImp(const TMIVectorImp<T,Alloc> &v, unsigned start, unsigned stop)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Public-Elementfunktionen

Current

```
T* Current();
```

Liefert einen Zeiger auf das aktuelle Objekt zurück.

Restart

```
void Restart();
```

Startet eine neue Iteration über den gesamten Vektor.

```
void Restart(unsigned start, unsigned stop);
```

Startet eine neue Iteration über den angegebenen Vektorbereich.

Operatoren

operator ++

```
const T& operator ++(int);
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das vor der Verschiebung aktuelle Objekt zurück (Post-Inkrement).

```
const T& operator ++();
```

Verschiebt die Iterator-Position auf das nächste Objekt und liefert das nach der Verschiebung aktuelle Objekt zurück (Prä-Inkrement).

operator int `operator int()`

Konvertiert den Iterator in einen Integerwert, um zu testen, ob noch Objekte im Iterator sind. Ist dies nicht der Fall, so ergibt die Konvertierung den Wert 0.

Template TIVectorImp vectimp.h

Implementiert einen Vektor von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Eine Beschreibung der Klassenelemente finden Sie unter *TMIVectorImp*.

Konstruktoren

`TIVectorImp(unsigned sz, unsigned d = 0);`

Erzeugt einen indirekten Vektor für *sz* Objektzeiger, die standardmäßig mit 0 initialisiert werden.

Template TIVectorIteratorImp vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von indirekten Vektoren mit eigener Speicherverwaltung. Siehe *TMIVectorIteratorImp*.

Konstruktoren

`TIVectorIteratorImp(const TIVectorImp<T> &v)`

Erzeugt ein Iterator-Objekt, mit dem *TIVectorImp-Objekte* durchlaufen werden können.

`TIVectorIteratorImp(const TIVectorImp<T> &v, unsigned l, unsigned u)`

Erzeugt ein Iterator-Objekt, mit dem *TIVectorImp-Objekte* durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TMICVectorImp vectimp.h

Implementiert einen abgezählten Vektor mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten.

Konstruktoren

`TMICVectorImp(unsigned sz, unsigned d = 0);`

Erzeugt einen indirekten abgezählten Vektor von Zeigern auf Objekte. *sz* gibt die Vektorgröße und *d* den Initialisierungswert an.

Public-Elementfunktionen

Neben den folgenden erbt *TMICVectorImp* die Elementfunktionen von TMIVectorImp. Siehe *TMIVectorImp*.

Add `int Add(T *t)`

Fügt dem Vektor einen Objektzeiger hinzu.

Find `unsigned Find (T *t) const;`

Sucht den angegebenen Objektzeiger und liefert seinen Indexwert zurück.

Protected-Elementfunktionen

Find `virtual unsigned Find (void *) const;`

Sucht den angegebenen Zeiger und liefert seinen Indexwert zurück.

Template TMICVectorIteratorImp vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von indirekten abgezählten Vektoren mit eigener Speicherverwaltung. Siehe *TMIVectorIterator* und *TMVectorIterator.*

Konstruktoren

`TMICVectorIteratorImp(const TMICVectorImp<T,Alloc> &v)`

Erzeugt ein Iterator-Objekt, mit dem *TMICVectorImp*-Objekte durchlaufen werden können.

`TMICVectorIteratorImp(const TMICVectorImp<T,Alloc> &v, unsigned l, unsigned u)`

Erzeugt ein Iterator-Objekt, mit dem *TMICVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TICVectorImp vectimp.h

Implementiert einen abgezählten Vektor von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Eine Beschreibung der Klassenelemente finden Sie unter *TMICVectorImp*.

Konstruktoren

```
TICVectorImp(unsigned sz, unsigned d = 0);
```

Erzeugt einen indirekten, abgezählten Vektor von Zeigern auf Objekte. *sz* gibt die Vektorgröße und *d* den Initialisierungswert an.

Template TICVectorIteratorImp vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von indirekten abgezählten Vektoren. Siehe *TMIVectorIterator* und *TMVectorIterator*.

Konstruktoren

```
TICVectorIteratorImp(const TICVectorImp<T> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TICVectorImp*-Objekte durchlaufen werden können.

```
TICVectorIteratorImp(const TMSVectorImp<T> &v, unsigned l, unsigned u)
```

Erzeugt ein Iterator-Objekt, mit dem *TICVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TMISVectorImp vectimp.h

Implementiert einen sortierten Vektor mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Eine Beschreibung der Klassenelemente finden Sie unter *TMICVectorImp*.

Konstruktoren

```
TMISVectorImp(unsigned sz, unsigned d = 0);
```

Erzeugt einen sortierten Vektor von Zeigern auf Objekte. *sz* gibt die Vektorgröße und *d* den Initialisierungswert an.

Template TMISVectorIteratorImp — vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von direkten sortierten Vektoren mit eigener Speicherverwaltung von Zeigern auf Objekte des Typs T. Siehe *TMIVectorIteratorImp* und *TMVectorIteratorImp*.

Konstruktoren

```
TMISVectorIteratorImp(const TMISVectorImp<T,Alloc> &v)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIVectorImp*-Objekte durchlaufen werden können.

```
TMISVectorIteratorImp(const TMIVectorImp<T,Alloc> &v, unsigned l,
                      unsigned u)
```

Erzeugt ein Iterator-Objekt, mit dem *TMIVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Template TISVectorImp — vectimp.h

Implementiert einen sortierten Vektor von Zeigern auf Objekte des Typs *T*. Da Zeiger stets über eine sinnvolle Kopiersemantik verfügen, kann diese Klasse jeden Objekttyp verwalten. Eine Beschreibung der Klassenelemente finden Sie unter *TMICVectorImp*.

Konstruktoren

```
TISVectorImp(unsigned sz, unsigned d = 0);
```

Erzeugt einen sortierten Vektor von Zeigern auf Objekte. *sz* gibt die Vektorgröße und d den Initialisierungswert an.

Template TISVectorIteratorImp — vectimp.h

Implementiert einen Vektor-Iterator zum Durchlaufen von indirekten sortierten Vektoren. Siehe *TMIVectorIteratorImp* und *TMVectorIteratorImp*.

Konstruktoren

```
TISVectorIteratorImp(const TISVectorImp<T> &v)
```
Erzeugt ein Iterator-Objekt, mit dem *TISVectorImp*-Objekte durchlaufen werden können.

```
TISVectorIteratorImp(const TISVectorImp<T> &v, unsigned start, unsigned stop)
```
Erzeugt ein Iterator-Objekt, mit dem *TISVectorImp*-Objekte durchlaufen werden können. Bei dieser Version kann ein Bereich angegeben werden.

Klasse TShouldDelete — shddel.h

TShouldDelete verwaltet den Besitzerstatus der indirekten Container. Das »Schicksal« der aus einem Container entfernten Elemente kann davon abhängig gemacht werden, ob der Container Besitzer seiner Elemente ist oder nicht. Analog kann auch beim Löschen eines Containers der Besitzerstatus bestimmen, ob die Elemente, die sich noch im aktuellen Gültigkeitsbereich befinden, ebenfalls gelöscht werden. Als virtuelle Basisklasse stellt *TShouldDelete* die Verwaltung des Besitzerstatus allen Containerklassen zur Verfügung. Die Elementfunktion *OwnsElements* kann zur Ermittlung oder Änderung des Container-Besitzerstatus verwendet werden. Mit der Elementfunktion *DelObj* kann man festlegen, ob Objekte im Container gelöscht werden sollen oder nicht.

Public-Datenelemente

DeleteType

```
enum DeleteType{ NoDelete, DefDelete, Delete };
```
Aufzählungswerte, die bestimmen, ob ein Objekt beim Entfernen aus dem Container gelöscht werden soll oder nicht.

Konstruktoren

```
TShouldDelete(DeleteType dt = Delete)
```
Erzeugt ein *TShouldDelete*-Objekt. Siehe Elementfunktion *DelObj*.

Public-Elementfunktionen

OwnsElements

`int OwnsElements()`

Liefert den Wert 1 zurück, falls der Container Besitzer seiner Elemente ist. Andernfalls wird 0 zurückgeliefert.

`void OwnsElements(int del)`

Ändert den Besitzerstatus wie folgt: ist *del* gleich 0, so ist der Container nicht Besitzer seiner Elemente. Ist *del* gleich 1, so ist der Container Besitzer seiner Elemente.

Protected-Elementfunktionen

DelObj

`int DelObj(DeleteType dt)`

Testet den Besitzerstatus und liefert den Wert 1 zurück, falls die Containerobjekte gelöscht werden sollen, bzw. den Wert 0, falls sie nicht gelöscht werden sollen. Die bestimmenden Faktoren sind der aktuelle Besitzerstatus und der Wert des Parameters *dt*, wie es in der folgenden Tabelle dargestellt wird.

	DelObj	
OwnsElements	**Ja**	**Nein**
NoDelete	Nein	Nein
DefDelete	Nein	Ja
Delete	Ja	Ja

DelObj liefert 1 zurück, falls »*dt* ist gleich *Delete*« oder »*dt* ist gleich *DefDelete* und der Container ist Besitzer seiner Elemente«. Daher liefert der *dt*-Wert *NoDelete* unabhängig vom Besitzerstatus den Wert 0 (nicht Löschen) zurück. Der *dt*-Wert *Delete* liefert unabhängig vom Besitzerstatus den Wert 1 (Löschen) zurück. Beim *dt*-Wert *DefDelete* wird 1 (Löschen) zurückgeliefert, falls der Container die Elemente besitzt, und 0 (nicht Löschen), falls er sie nicht besitzt.

K A P I T E L 8

Die mathematischen C++ Klassen

In diesem Kapitel wird die Mathematikunterstützung von Borland C++ vorgestellt, die auf C++ Klassen basiert. Die darin enthaltenen mathematischen Operationen sind nur in C++ Programmen verfügbar. Ein C++ Programm, das diese Klassen, die von ihnen definierten numerischen Typen oder irgendeine ihrer Friend- oder Elementfunktionen einsetzt, kann jedoch alle normalen ANSI C Mathematikroutinen benutzen.

Die beiden Klassen *bcd* und *complex* realisieren numerische Datentypen. Neben diesen Datentypen definiert jede Klasse Funktionen, mit denen Operationen mit den jeweiligen Typen ausgeführt werden können (beispielsweise Konvertierung nach und von den Typen *bcd* und *complex*). Alle notwendigen Operatoren werden von beiden Klassen überladen.

Die mathematischen Klassen sind in keine Hierarchie eingebettet. Beide binden jedoch die Header-Datei iostream.h ein.

Die Portabilität der Klassen *bcd* und *complex* ist in der folgenden Tabelle zu sehen:

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Klasse bcd — bcd.h

Die Konstruktoren der Klasse *bcd* erzeugen binär codierte Dezimalzahlen (BCD) auf Integer- oder Gleitkommawerten. Die auf Seite 495 beschriebene Friend-Funktion *real* konvertiert *bcd*-Zahlen in **long double-**Werte

Wenn Sie *bcd*-Zahlen erzeugt haben, dann können Sie diese in Ausdrücken beliebig mit **int-, double-** oder anderen numerischen Typen mi-

schen. Sie können *bcd*-Zahlen auch in allen normalen ANCI C Mathematikfunktionen einsetzen.

Die folgenden ANSI Mathematikfunktionen sind überladen, so daß sie mit *bcd*-Zahlen zusammenarbeiten:

```
friend bcd  abs(bcd &);
friend bcd  acos(bcd &);
friend bcd  asin(bcd &);
friend bcd  atan(bcd &);
friend bcd  cos(bcd &);
friend bcd  cosh(bcd &);
friend bcd  exp(bcd &);
friend bcd  log(bcd &);
friend bcd  log10(bcd & base, bcd & expon);
friend bcd  pow(bcd &);
friend bcd  sin(bcd &);
friend bcd  sinh(bcd &);
friend bcd  sqrt(bcd &);
friend bcd  tan(bcd &);
friend bcd  tanh(bcd &);
```

Die Beschreibung dieser Funktionen finden Sie in Kapitel 3.

Die Klasse *bcd* überlädt zudem die Operatoren **+, -, *, /, +=, -=, *=, /=, =, ==** und **!=.** Diese Operatoren realisieren die arithmetische Verknüpfung von *bcd*-Zahlen im üblichen Sinne.

Die Operatoren **<<** und **>>** für Stream-Ein-/Ausgabe von *bcd*-Werten sind ebenso überladen, wie dies in iostream.h für andere Datentypen geschieht.

bcd-Zahlen besitzen eine Genauigkeit von 17 Dezimalziffern und einen Wertebereich von 1×10^{-125} bis 1×10^{125}.

Die Zahlen werden auf die nächste ganze Zahl gerundet – ist diese nicht eindeutig, so wird auf die nächste gerade Zahl gerundet.

Public-Konstruktoren

```
bcd();
```

Der Standardkonstruktor, der normalerweise zur Deklaration einer Variablen vom Typ *bcd* verwendet wird.

```
bcd i;       // Erzeugt eine bcd-Zahl
bcd j = 37;  // Erzeugt und initialisiert eine bcd-Zahl
```

```
bcd(int x);
```

Dieser Konstruktor definiert eine *bcd*-Variable durch eine Integervariable oder direkt durch einen Integerwert.

```
int i = 15;
bcd j = bcd(i);    // Initialisiert j mit einem zuvor deklarierten Typ
bcd k = bcd(12);   // Erzeugt k mit dem angegebenen Integerwert
```

Das obige Beispiel erzeugt die folgenden Variablen:

```
i = 15    j = 15    k = 12
```

`bcd(unsigned int x);`

Dieser Konstruktor erzeugt eine *bcd*-Variable aus einem **unsigned int-**Wert, der dem Konstruktor als Variable oder direkt als Wert übergeben werden kann.

`bcd(long x);`

Dieser Konstruktor erzeugt eine *bcd*-Variable aus einer **long-**Variable oder direkt aus einem **long**-Wert.

`bcd(unsigned long x);`

Dieser Konstruktor erzeugt eine *bcd*-Variable aus einem **unsigned long-**Wert.

`bcd(double x, int decimals = Max);`

Dieser Konstruktor erzeugt eine *bcd*-Variable aus einer Gleitkommavariablen des Typs **double.** Der Konstruktor kann auch direkt einen **double**-Wert übernehmen.

Eine vom Standard abweichende Genauigkeit der Darstellung (die Anzahl der Ziffern nach dem Dezimalpunkt) kann wie im folgenden Beispiel über den Parameter *decimals* spezifiziert werden:

```
double x = 1.2345;   // Übliche Deklaration und Initialisierung
bcd y = bcd(x, 2);   // Erzeugung einer bcd-Variable aus x
```

Da die Genauigkeit für *y* auf 2 gesetzt ist, wird *y* mit 1.23 initialisiert.

`bcd(long double x, int decimals = Max);`

Dieser Konstruktor erzeugt eine *bcd*-Variable aus einer Gleitkommavariablen des Typs **long double.** Der Konstruktor kann auch direkt einen **long double-**Wert übernehmen.

Eine vom Standard abweichende Genauigkeit der Darstellung (die Anzahl der Ziffern nach dem Dezimalpunkt) kann über den Parameter *decimals* spezifiziert werden.

Friend-Funktionen

real

```
long double real(bcd number);
```

Die Funktion *real* konvertiert eine *bcd*-Zahl zurück in einen **long double**-Wert. Im Kapitel2 des *Programmierhandbuch* finden Sie weitere Erläuterungen zu arithmetischen Konvertierungen.

Klasse complex complex.h

Die Klasse *complex* enthält komplexe Zahlen. Wenn Sie komplexe Zahlen erzeugt haben, können Sie diese in Ausdrücken beliebig mit **int-, double-** oder anderen numerischen Werten mischen. Sie können *complex*-Zahlen auch in allen normalen ANCI C Mathematikfunktionen einsetzen.

Die Klasse *complex* überlädt zudem die Operatoren **+, -, *, /, +=, -=, *=, /=, =, ==** und **!=.** Diese Operatoren realisieren die arithmetische Verknüpfung von komplexen Zahlen im üblichen Sinne.

Die Operatoren **<<** und **>>** für Stream-Ein-/Ausgabe von *bcd*-Werten sind ebenso überladen, wie dies in iostream.h für andere Datentypen geschieht.

Wenn Sie nicht in C++ sondern in C programmieren wollen, so stehen Ihnen nur die Struktur *complex* und die Routine *cabs* zur Verfügung, die den Absolutbetrag einer komplexen Zahl ermittelt. **struct** *complex* und *cabs* sind in math.h definiert.

Public-Konstruktoren

```
complex();
```

Der Standardkonstruktor, der normalerweise zur Deklaration einer Variablen vom Typ *complex* verwendet wird.

```
complex i;       // Erzeugt eine komplexe Zahl
complex j = 37;  // Erzeugt und initialisiert eine komplexe Zahl
```

```
complex(double real, double imag = 0);
```

Erzeugt einen *complex*-Wert aus einem **double-**Wert. Es wird der Real- und der Imaginärteil angegeben. Fehlt der Imaginärteil, so wird *imag* auf den Wert 0 gesetzt.

Friend-Funktionen

abs

```
friend double abs(complex& val);
```

Liefert den Absolutwert einer komplexen Zahl.

Die komplexe Version von *abs* liefert einen **double-**Wert zurück. Alle übrigen mathematischen Funktionen liefern einen *complex*-Typ, wenn ihnen ein komplexer Wert übergeben wird.

acos

```
friend complex acos(complex& z);
```

Berechnet den Arcuscosinus des Arguments.

Der komplexe Arcuscosinus ist folgendermaßen definiert:

$$acos(z) = -i * log(z + i * sqrt(1 - z^2))$$

arg

```
double arg(complex z);
```

arg ermittelt den Winkel der komplexen Zahl z in der komplexen Ebene in der Einheit *rad.*

Die positive reelle Achse hat den Winkel 0, die positive imaginäre Achse den Winkel *pi*/2. Wird der Funktion der Wert 0 übergeben, so liefert sie 0 zurück.

arg(x) liefert *atan2(imag(x),real(x))* zurück.

asin

```
friend complex asin(complex& z);
```

Berechnet den Arcussinus.

Der komplexe Arcussinus ist folgendermaßen definiert:

$$asin(z) = -i * log(i * z + sqrt(1 - z^2))$$

atan

```
friend complex atan(complex& z);
```

Berechnet den Arcustangens des Arguments.

Der komplexe Arcustangens ist folgendermaßen definiert:

```
atan(z) = -0.5 * i * log((1 + i * z)/(1 - i * z))
```

conj

```
complex conj(complex z);
```

Berechnet den konjungierten Wert einer komplexen Zahl. `conj(z)` entspricht `complex(real(z), -imag(z))`.

cos

```
friend complex cos(complex& z);
```

Berechnet den Cosinus des Arguments.

Der komplexe Cosinus ist folgendermaßen definiert:

```
cos(z) = (exp(i * z) + exp(-i * z)) / 2
```

cosh

```
friend complex cosh(complex& z);
```

Berechnet den Cosinus hyperbolicus der Arguments.

Der komplexe Cosinus hyperbolicus ist folgendermaßen definiert:

```
cosh(z) = (exp(z) + exp(-z)) / 2
```

exp

```
friend complex exp(complex& z);
```

Berechnet *e* hoch *z*.

Die komplexe Exponentialfunktion ist folgendermaßen definiert:

```
exp(x + i * y) = exp(x) (cos(y) + i * sin(y))
```

imag

```
double imag(complex z);
```

Liefert den Imaginärteil einer komplexen Zahl zurück.

Die zu einer komplexen Zahl gehörenden Daten bestehen aus zwei **double**- Gleitkommawerten. *imag* liefert den Imaginäranteil der komplexen Zahl zurück.

log

```
friend complex log(complex& z);
```

Berechnet den natürlichen Logarithmus von *z*.

Der komplexe natürliche Logarithmus ist folgendermaßen definiert:

```
log(z) = (log(abs(z)) + i * arg(z)
```

log10

```
friend complex log10(complex& z);
```

Berechnet $\log_{10}(z)$.

Der allgemeine komplexe Logarithmus ist folgendermaßen definiert:

```
log10(z) = log(z) / log(10)
```

norm

```
double norm(complex x);
```

Berechnet das Quadrat des Absolutbetrags. *norm(x)* liefert den Wert *real(x) * real(x) + imag(x) * imag(x)* zurück.

Beim Aufruf von *norm* kann ein Überlauf entstehen, falls der Real- oder der Imaginärteil genügend groß ist.

polar

```
complex polar(double mag, double angle = 0);
```

Liefert einen komplexen Wert für einen gegebenen Betrag und Winkel.

polar(mag, angle) entspricht *complex(mag * cos(angle), mag * sin(angle)).*

pow

```
friend complex pow(complex& base, double expon);
friend complex pow(double base, complex& expon);
friend complex pow(complex& base, complex& expon);
```

Berechnet *base* hoch *expon*.

Die komplexe Funktion *pow* ist folgendermaßen definiert:

```
pow(base, expon) = exp(expon * log(base))
```

real

```
double real(complex z);
```

Die Funktion *real* konvertiert die komplexe Zahl *z* zurück in einen **double**-Wert. Die Friend-Funktion liefert den Realteil der komplexen Zahl zurück oder konvertiert einen *complex*-Wert in einen **double**-Wert. Die zu einer komplexen Zahl gehörenden Daten bestehen aus zwei **double**-Gleitkommawerten. *real* liefert den Realteil zurück.

Im Kapitel 2 des *Programmierhandbuch* finden Sie weitere Erläuterungen zu arithmetischen Konvertierungen.

sin

```
friend complex sin(complex& z);
```

Berechnet den Sinus des Arguments.

Der komplexe Sinus ist folgendermaßen definiert:

```
sin(z) = (exp(i * z) - exp(-i * z)) / (2 * i)
```

sinh

```
friend complex sinh(complex& z);
```

Berechnet den Sinus hyperbolicus des Arguments.

Der komplexe Sinus hyperbolicus ist folgendermaßen definiert:

```
sinh(z) = (exp(z) - exp(-z)) / 2
```

sqrt

```
friend complex sqrt(complex& x);
```

Berechnet die positive Quadratwurzel des Arguments x.

Der Winkel der mit *sqrt* berechneten Quadratwurzel *x* in der komplexen Ebene beträgt *arg(x)*/2. Die komplexe Quadratwurzel ist folgendermaßen definiert:

```
sqrt(x) = sqrt(abs(x)) * (cos(arg(x) / 2) + i * sin(arg(x) / 2))
```

tan

```
friend complex tan(complex& z);
```

Berechnet den Tangens des Arguments.

Der komplexe Tangens ist folgendermaßen definiert:

```
tan(z) = sin(z) / cos(z)
```

tanh

```
friend complex tanh(complex& z);
```

Berechnet den Tangens hyperbolicus des Arguments.

Der komplexe Tangens hyperbolicus ist folgendermaßen definiert:

```
tanh(z) = sinh(z) / cosh(z)
```

K A P I T E L 9

Diagnose-Makros

Borland stellt Ihnen eine Reihe von Makros für das Debuggen von C++ Programmen zur Verfügung. Diese in checks.h definierten Makros können unter Windows und unter DOS eingesetzt werden. Es gibt zwei Makrotypen: Standardmakros und erweiterte Makros. Die Standardmakros umfassen:

- CHECK
- PRECONDITION
- TRACE
- WARN

Zu den erweiterten Makros gehören:

- CHECKX
- PRECONDITIONX
- TRACEX
- WARNX

Die Standardmakros ermöglichen einfache Wertüberprüfungen und Meldungsausgaben. Mit den erweiterten Makros können Sie Makrogruppen bilden, die Sie einzeln aktivieren bzw. deaktivieren können. Weiterhin können Sie bestimmte Makros innerhalb einer Gruppe über einen numerischen Schwellenwert aktivieren bzw. deaktivieren.

Drei Präprozessorsymbole steuern die Expansion der Diagnose-Makros: __DEBUG, __TRACE und __WARN. Nur wenn eines dieser Symbole während der Compilierung definiert ist, werden die zugehörigen Makros expandiert und der Diagnose-Code erzeugt. Die Symbole können Sie in der Kommandozeile mit dem Schalter **-D** oder im Quelltext durch `#define` Anweisungen definieren.

Die Diagnose-Makros werden gemäß folgender Tabelle aktiviert:

Diagnose-Makros

	__DEBUG=1	__DEBUG=2	__TRACE	__WARN
PRECONDITION	X	X		
PRECONDITIONX	X	X		
CHECK		X		
CHECKX		X		
TRACE			X	
TRACEX			X	
WARN				X
WARNX				X

Um eine Diagnose-Version Ihres Programms zu erzeugen, setzen Sie die Diagnose-Makros an wichtige Stellen innerhalb Ihres Programms und compilieren es mit den entsprechenden Präprozessor-Symbolen. Diagnose-Versionen der Borland-Klassenbibliotheken sind in ähnlicher Weise aufgebaut.

In den folgenden Abschnitten werden die Standard- und die erweiterten Diagnose-Makros beschrieben. Es werden Beispiele zu ihrer Verwendung vorgestellt. Zwei weitere Abschnitte befassen sich mit der Ausgabe von Meldungen und der Steuerung von Makrogruppen zur Laufzeit.

Standard-Diagnose-Makros — checks.h

CHECK

`CHECK(<cond>)`

<*cond*> wird ausgegeben und eine Exception ausgelöst, falls <*cond*> gleich 0 ist. Mit dem Makro CHECK überprüfen Sie Werte innerhalb einer Funktion.

PRECONDITION

`PRECONDITION(<cond>)`

<*cond*> wird ausgegeben und eine Exception ausgelöst, falls <*cond*> gleich 0 ist. Verwenden Sie das Makro PRECONDITION zur Gültigkeitsüberprüfung von Funktionsargumenten und für alle anderen Tests, die den korrekten Aufruf einer Funktion sicherstellen sollen.

TRACE

`TRACE(<msg>)`

Gibt <*msg*> aus. TRACE wird zur Ausgabe von allgemeinen Meldungen benutzt, die nicht von einer Bedingung abhängen.

WARN

`WARN(<cond>,<msg>)`

Ausgabe von <*msg*> falls <*cond*> ungleich 0 ist. WARN wird zur Ausgabe von bedingten Meldungen verwendet.

Ein Beispiel Das folgende Programm verdeutlicht den Einsatz der Standardmakros TRACE und WARN:

```
#include <checks.h>

int main()
{
   TRACE("Hallo Welt!");
   WARN(5 != 5, "Ungleich-Operator defekt!");
   WARN(5 != 7, "Ungleich-Operator funktioniert!");

   return 0;
}
```

Wird der obige Quelltext mit definierten Symbolen __TRACE und __WARN compiliert, so werden zur Laufzeit folgende Meldungen erzeugt:

```
Trace PROG.C 5: [Def] Hallo Welt!
Warning PROG.C 7: [Def] Ungleich-Operator funktioniert!
```

Die obige Ausgabe besagt, daß die Meldung »Hallo Welt!« vom Standardmakro TRACE in der Zeile 5 und die Meldung »Ungleich-Operator funktioniert!« vom Standardmakro WARN in Zeile 7 von PROG.C ausgegeben wurde.

Die Standardmakros werden einfach zu den erweiterten Diagnose-Makros expandiert, indem die Makrogruppe auf »Def« und der Schwellenwert auf 0 gesetzt wird.

Erweiterte Diagnose-Makros — checks.h

Die Makros CHECKX und PRECONDITIONX erweitern CHECK und PRECONDITION, so daß zusätzlich eine Meldung ausgegeben wird, falls die Bedingung nicht zutrifft.

Die Makros TRACEX und WARNX erweitern TRACE und WARN, so daß Sie Makrogruppen anlegen können, die einzeln aktivierbar bzw. deaktivierbar sind. TRACEX und WARNX benötigen zusätzliche Argumente, mit denen Sie die Gruppenzugehörigkeit und den Schwellenwert festlegen können, bei dem das Makro ausgeführt werden soll. Das Makro wird nur dann ausgeführt, wenn seine Makrogruppe aktiviert ist und einen Schwellenwert besitzt, der nicht kleiner als das Schwellenwert-Argument des Makros selbst ist.

Im folgenden Abschnitt werden die erweiterten Diagnose-Makros beschrieben.

CHECKX `CHECKX(<cond>,<msg>)`

<*msg*> wird ausgegeben und eine Exception ausgelöst, falls <*cond*> gleich 0 ist. Mit dem Makro CHECKX überprüfen Sie Werte innerhalb einer Funktion.

PRECONDITIONX `PRECONDITIONX(<cond>,<msg>)`

<*msg*> wird ausgegeben und eine Exception ausgelöst, falls <*cond*> gleich 0 ist. Verwenden Sie das Makro PRECONDITIONX zur Gültigkeitsüberprüfung von Funktionsargumenten und für alle anderen Tests, die den korrekten Aufruf einer Funktion sicherstellen sollen.

TRACEX `TRACEX(<group>,<level>,<msg>)`

Wird nur ausgeführt, wenn <*group*> und <*level*> aktiviert sind.

WARNX `WARNX(<group>,<cond>,<level>,<msg>)`

Wird nur ausgeführt, wenn <*group*> und <*level*> aktiviert sind.

Für die Verwendung von TRACEX und WARNX müssen Sie Makrogruppen definieren. Die folgenden Makros erzeugen Makrogruppen:

DIAG_DECLARE_GROUP `DIAG_DECLARE_GROUP(<name>)`

Deklariert die Makrogruppe <*name*>.

DIAG_DEFINE_GROUP `DIAG_DEFINE_GROUP(<name>,<enabled>,<level>)`

Definiert die Gruppe <*name*>.

DIAG_CREATE_GROUP `DIAG_CREATE_GROUP(<name>,<enabled>,<level>)`

Deklariert und definiert die Gruppe <*name*>.

Mit den folgenden Makros kann auf Makrogruppen zugegriffen werden:

DIAG_ENABLE `DIAG_ENABLE(<group>,<state>)`

Entsprechend dem Inhalt von <*state*> wird die Gruppe aktiviert bzw. deaktiviert.

DIAG_ISENABLED `DIAG_ISENABLED(<group>)`

Liefert einen Wert ungleich null, falls die Makrogruppe <*group*> deaktiviert ist.

Mit den folgenden Makros kann auf die Schwellenwerte zugegriffen werden:

DIAG_SETLEVEL

```
DIAG_SETLEVEL(<group>,<level>)
```

Setzt den Schwellenwert von <*group*> auf den Wert <*level*>.

DIAG_GETLEVEL

```
DIAG_GETLEVEL(<group>)
```

Liefert den Schwellenwert von <*group*>.

Schwellenwerte sind beliebige numerische Werte, über die Makros aktiviert werden können. Ein Makro, dessen Schwellenwert größer als der seine Gruppe ist, wird nicht ausgeführt. Hat eine Makrogruppe beispielsweise den Schwellenwert 0 (der Standardwert), so werden alle Makros ignoriert, die zwar zur Gruppe gehören, aber einen Schwellenwert von 1 oder größer besitzen.

Ein Beispiel Im folgenden Quelltext PROG.C werden die beiden Makrogruppen *Foo* und *Bar* definiert, die dann als Argumente in den erweiterten Diagnose-Makros verwendet werden:

```
#include<checks.h>

DIAG_CREATE_GROUP(Foo,1,0);
DIAG_CREATE_GROUP(Foo,1,0);

int main(int argc, char **argv)
{
   TRACE("Funktioniert immer, argc=" << argc);

   TRACEX(Foo, 0, "Hallo");
   TRACEX(Bar, 0, "Hallo");

   DIAG_DISABLE(Foo);

   TRACEX(Foo, 0, "Wird nicht ausgeführt - Gruppe ist deaktiviert!");
   TRACEX(Bar, 0, "Wird nicht ausgeführt - Schwellenwert zu groß!");

   return 0;
}
```

Wird der obige Quelltext bei definiertem Symbol __TRACE compiliert, so wird zur Laufzeit folgende Ausgabe erzeugt:

```
Trace PROG.C 8: [Def] Funktioniert immer, argc=1
Trace PROG.C 10: [Foo] Hallo
Trace PROG.C 11: [Bar] Hallo
```

Beachten Sie, daß die beiden letzten Makros nicht ausgeführt werden. Im ersten Fall ist die Gruppe *Foo* deaktiviert, im zweiten ist der Schwellen-

wert des Makros (3) größer als der Schwellenwert der Makrogruppe *Bar* (0).

Ausgabe von Meldungen

Das den Makros CHECKX, PRECONDITIONX, TRACE, TRACEX, WARN und WARNX übergebene Argument <*msg*>, wird – sofern die zugehörige Bedingung erfüllt ist – in einen Ausgabe-Stream eingefügt. Somit können auch mehrere Objekte in den Ausgabe-Stream eingefügt werden (beispielsweise `TRACE("Maus @ " << x << "," << y;).` Die Verwendung von Streams ermöglicht eine Erweiterung auf verschiedene Objekttypen und die Benutzung von Parametern innerhalb der Meldungen.

Die von den Makros erzeugten Diagnosemeldungen können zur Laufzeit angesehen werden. Ist die Zielumgebung Windows, dann werden die Ausgaben der Funktion *OutputDebugString* übergeben und können mit den Hilfsprogrammen DBWIN.EXE oder UX.SYS betrachtet werden. Im Turbo Debugger erscheinen die Ausgaben im Log-Fenster. Unter DOS werden die Ausgaben nach *stdout* geschrieben und können in der Kommandozeile leicht umgelenkt werden.

Makrosteuerung zur Laufzeit

Die Diagnose-Gruppen können zur Laufzeit mit den vorgestellten Steuerungsmakros oder durch direkte Änderung der Gruppeninformationen innerhalb des Turbo Debugger beeinflußt werden.

Die Gruppeninformationen sind in der Template-Klasse *TDiagGroup*<*TDiagGroupClass##Group*> enthalten, wobei *##Group* den Namen der Makrogruppe bezeichnet. Zur Klasse gehört die statische Struktur *Flags*, die das aktiv-Flag und den Schwellenwert enthält. Um beispielsweise die Gruppe *Foo* zu aktivieren, müssen Sie die Variable *TDiagGroup*<*TDiagGroupClassFoo*>*::Flags.Enabled* auf 1 setzen.

K A P I T E L 10

Laufzeitunterstützung

In diesem Kapitel werden in alphabetischer Reihenfolge alle Funktionen und Klassen zur Laufzeitunterstützung detailliert beschrieben. Alle Klassenoperatoren und Elementfunktionen werden unmittelbar nach dem Klassenkonstruktor aufgeführt. Im Kapitel 3 des *Programmierhandbuch* werden die Schlüsselwörter zur Exception-Behandlung erläutert.

Die Portabilität aller Klassen und Funktionen in diesem Kapitel sieht folgendermaßen aus:

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
■	■	■	■	■	■	■

Klasse Bad_cast — typeinfo.h

Wenn **dynamic_cast** bei einer versuchten Typumwandlung auf eine Referenz scheitert, dann kann der Ausdruck die Exception *Bad_cast* auslösen. Beachten Sie, daß bei einer gescheiterten Typumwandlung auf einen Zeigertyp der Ergebniszeiger ein Nullzeiger ist.

Klasse Bad_typeid — typeinfo.h

Wenn der Operand von **typeid** ein dereferenzierter Nullzeiger ist, dann kann der Operator **typeid** die Exception *Bad_typeid* auslösen.

Funktion set_new_handler — new.h

```
typedef void (new * new_handler)() throw(xalloc);
new_handler set_new_handler(new_handler my_handler);
```

set_new_handler installiert eine Funktion, die jedesmal dann aufgerufen wird, wenn der globale Operator **new()** oder **new[]()** den angeforderten Speicherplatz nicht reservieren konnte. Standardmäßig lösen die **new**-Operatoren eine *xalloc*-Exception aus, wenn Speicherplatz nicht reserviert werden kann. Dieses Standardverhalten können Sie durch einen Aufruf von *set_new_handler* ändern, mit dem eine neue Behandlungsroutine festgelegt wird. Durch einen Aufruf von *set_new_handler(0)* wird das traditionelle **new-**Verhalten wiederhergestellt, bei dem keine Exception ausgelöst wird.

Kann der angeforderte Speicherplatz von **new** nicht reserviert werden, so wird die vom vorherigen *set_new_handler*-Aufruf installierte Behandlungsroutine aufgerufen. **new** liefert den Wert 0 zurück, falls keine solche Routine von *set_new_handler* installiert wurde. In *my_handler* sollten die Maßnahmen enthalten sein, die bei einer gescheiterten Speicherreservierung unternommen werden müssen. Der in new.h definierte Typ *new_handler* beschreibt eine Funktion ohne Argumente und dem Rückgabetyp **void.** Eine *new_handler*-Funktion kann die Exception *xalloc* auslösen.

Die benutzerdefinierte Funktion *my_handler* sollte eine der folgenden Maßnahmen durchführen:

- Rücksprung nach Freigabe des Speichers
- Auslösen einer *xalloc*-Exception oder einer davon abgeleiteten Exception
- Aufruf von *abort* oder *exit*

Kehrt *my_handler* zurück, dann wird **new** erneut versuchen, die Speicheranforderung zu erfüllen.

Im Idealfall gibt *my_handler* Speicher wieder frei und kehrt dann zurück. **new** kann daraufhin den angeforderten Speicher reservieren und das Programm kann fortgesetzt werden. Wenn *my_handler* den angeforderten Speicher jedoch nicht für **new** freigeben kann, dann muß die Funktion eine Exception auslösen oder das Programm beenden, da ansonsten eine Endlosschleife entsteht.

Vorzugsweise sollten Sie jedoch die Operatoren **new()** und **new[]()** überladen, um die geeigneten Maßnahmen in Ihren Anwendungen treffen zu können.

set_new_handler liefert die alte Behandlungsroutine zurück, sofern eine installiert wurde.

Die als Argument übergebene, benutzerdefinierte Funktion *my_handler* sollte keinen Wert zurückgeben.

Siehe auch die Beschreibungen zu *abort*, *exit* und *new_handler* (globale Variable).

Funktion set_terminate **except.h**

```
typedef void (*terminate_function)();
terminate_function set_terminate(terminate_function t_func);
```

Mit Hilfe von *set_terminate* können Sie eine Funktion installieren, die das Verhalten bei Programmende für den Fall bestimmt, daß keine Behandlungsroutine für eine Exception gefunden werden konnte. Die durchzuführenden Aktionen sind in *t_func* definiert, das als eine Funktion des Typs *terminate_function* deklariert ist. Der in except.h definierte Funktionstyp *terminate_function* beschreibt eine Funktion ohne Argumente und mit dem Rückgabetyp **void.**

Eine Exception, für die keine Behandlungsroutine gefunden wird, führt standardmäßig zum Aufruf der Funktion *terminate*, die ihrerseits wiederum die Funktion *abort* aufruft. Das Programm wird dann mit der Meldung `Abnormal program termination` abgebrochen. Wenn Sie möchten, daß anstelle von *abort* eine andere Funktion von *terminate* aufgerufen wird, dann sollten Sie Ihre eigene *t_func*-Funktion definieren. Mit *set_terminate* wird Ihre *t_func-Funktion* anstelle von *terminate* installiert. Sie können auf diese Weise alle denkbaren Programmaktionen implementieren, die von *abort* nicht durchgeführt werden.

set_terminate liefert die zuvor installierte Funktion zur Programmbeendigung zurück.

In der Funktion *t_func* muß das Programm beendet werden. Die benutzerdefinierte Funktion darf nicht zur aufrufenden Funktion – der Funktion *terminate* – zurückkehren. Wird dies dennoch versucht, so ist das folgende Programmverhalten undefiniert. Ebenso ist das Auslösen einer Exception in *t_func* ein Fehler.

Siehe auch die Beschreibungen von *abort*, *set_unexpected* und *terminate*.

Funktion set_unexpected **except.h**

```
typedef void ( * unexpected_function )();
unexpected_function set_unexpected(unexpected_function unexpected_func);
```

Mit Hilfe von *set_unexpected* können Sie eine Funktion installieren, die beim Auftreten einer Exception aufgerufen wird, die in der Exception-Beschreibung der auslösenden Funktion nicht aufgeführt ist. Die durchzuführenden Maßnahmen sind in der Funktion *unexpected_func* definiert, die als Funktion des Typs *unexpected_function* deklariert ist. Der Funktions-typ *unexpected_function* beschreibt eine Funktion ohne Argumente und mit dem Rückgabetyp **void**.

Beim Auftreten einer unerwarteten Exception wird standardmäßig die Funktion *unexpected* aufgerufen. Wurde die Funktion *unexpected_func* definiert, so wird sie von *unexpected* aufgerufen und die Programmsteurung geht an die benutzerdefi- nierte Funktion *unexpected_func* über. Andernfalls wird *terminate* aufgerufen.

set_unexpected liefert die zuvor installierte Funktion zurück.

Die benutzerdefinierte Funktion *unexpected_func* darf nicht zur aufrufenden Funktion – der Funktion *unexpected* – zurückkehren. Wird dies dennoch versucht, so ist das folgende Programmverhalten undefiniert.

unexpected_func kann ebenfalls die Funktion *abort*, *exit* oder *terminate* aufrufen.

Siehe auch die Beschreibungen zu *abort*, *exit*, *set_terminate* und *terminate*.

Funktion terminate — except.h

```
void terminate();
```

Die Funktion *terminate* kann von *unexpected* oder vom Programm aufgerufen werden, wenn für eine Exception keine Behandlungsroutine gefunden werden kann. Die von *terminate* durchgeführte Standardaktion ist der Aufruf von *abort*, was den sofortigen Programmabbruch zur Folge hat.

Sie können das Programmverhalten beim Auftreten einer unbekannten Exception ändern. Wenn Sie nicht wollen, daß Ihr Programm durch einen Aufruf von *abort* abgebrochen wird, dann können Sie eine Funktion definieren, die stattdessen aufgerufen wird. Eine solche Funktion (vom Typ *terminate_function*) wird von *terminate* aufgerufen, sofern sie zuvor mit *set_terminate* installiert wurde.

Die Funktion *terminate* kehrt nicht zurück.

Siehe auch die Beschreibungen zu *abort* und *set_terminate*.

Klasse Type_info — typeinfo.h

Stellt Typinformationen zur Verfügung.

Public-Konstruktoren

Keiner.

Es gibt lediglich einen private-Konstruktor. Sie können *Type_info*-Objekte daher nicht selbst erzeugen. Deklarieren Sie Ihre Objekte als **__rtti**-Typen

oder verwenden Sie den Compiler-Schalter **-RT**, und der Compiler erzeugt Ihre Objekte mit den Elementen von *Type_info*.

Type_info-Referenzen werden mit dem Operator **typeid** generiert. Im Kapitel 2 des Programmierhandbuch finden Sie eine Beschreibung von **typeid.**

Operatoren

Operator ==

```
int operator==(const Type_info &) const;
```

Ermöglicht Vergleiche von Typinformationen.

Operator !=

```
int operator!=(const Type_info &) const;
```

Ermöglicht Vergleiche von Typinformationen.

Public-Elementfunktionen

before

```
int before(const Type_info &);
```

Mit dieser Funktion können Sie die lexikalische Ordnung zweier Typen vergleichen. Um beispielsweise die Typen ***T1*** und ***T2*** zu vergleichen, verwenden Sie folgende Syntax:

```
typeid( T1 ).before(typid( T2 ));
```

Die Funktion *before* liefert 0 oder 1 zurück.

fname
name

```
const char* __far fname() const;
const char* name() const;
```

Die Funktionen *fname* und *name* sind in ihrer Funktionalität identisch. *fname* wird im Speichermodell Large verwendet.

Beide Funktionen liefern einen druckbaren String zurück, der den Typnamen des **typeid**-Operanden identifiziert. Der Speicherplatz des zugehörigen String wird bei jedem Funktionsaufruf überschrieben.

Funktion unexpected — except.h

```
void unexpected();
```

Die Funktion *unexpected* wird aufgerufen, wenn eine Funktion eine Exception auslöst, die in der Exception-Beschreibung nicht aufgeführt ist. Das Programm ruft in diesem Fall die Funktion *unexpected* auf, die ihrerseits eine benutzerdefinierte und mit *set_unexpected* installierte Funktion aufruft. Ist keine solche Funktion mit *set_unexpected* installiert worden, dann wird die Funktion *terminate* aufgerufen.

Die Funktion *unexpected* kehrt nicht zurück, sie kann jedoch eine Exception auslösen.

Siehe auch die Beschreibungen zu *set_unexpected* und *terminate*.

Klasse xalloc — except.h

Meldet Fehler bei Speicherreservierungen.

Public-Konstruktoren

```
xalloc( const string &msg, size_t size );
```

Die Klasse *xalloc* besitzt keinen Standardkonstruktor. Bei jeder Verwendung von *xalloc* muß eine Meldung definiert werden, die ausgegeben wird, wenn eine Speicherreservierung der Größe *size* nicht erfüllt werden kann. Der Typ *string* ist in der Header-Datei cstring.h definiert.

Public-Elementfunktionen

raise

```
void raise() throw(xalloc);
```

Der Aufruf von *raise* führt dazu, daß eine *xalloc*-Exception ausgelöst wird. Tatsächlich wird ***this** ausgelöst.

requested

```
size_t requested() const;
```

Liefert die Byte-Anzahl zurück, die bei der Speicherreservierung angefordert wurden.

Klasse xmsg — except.h

Ausgabe einer Meldung zu einer Exception.

Public-Konstruktoren

```
xmsg( string msg );
```

Es gibt keinen Standardkonstruktor für *xmsg*. Für jedes *xmsg*-Objekt muß eine *string*-Meldung explizit angegeben werden. Der Typ *string* ist in der Header-Datei cstring.h definiert.

Public-Elementfunktionen

raise

```
void raise() throw(xmsg);
```

Der Aufruf von *raise* führt zum Auslösen einer *xmsg*-Exception. Tatsächlich wird ***this** ausgelöst.

why

```
string why() const;
```

Liefert den zu *xmsg* gehörende Meldungsstring zurück. Da für jedes *xmsg*-Objekt eine Meldung explizit angegeben werden muß, sollte jede Instanz eine eindeutige Meldung besitzen.

K A P I T E L 11

Die C++ Utility-Klassen

In diesem Kapitel werden die folgenden Klassen vorgestellt:

- Datumsklasse (date.h)
- Dateiklassen (file.h)
- Stringklassen (cstring.h)
- Thread-Klassen (thread.h)
- Zeitklassen (time.h)

Die zu diesen Klassen gehörenden Header-Dateien finden Sie entweder in \BC4\INCLUDE oder in \BC4\INCLUDE\CLASSLIB.

Klasse TDate date.h

```
class TDate
```

Die Klasse TDate beinhaltet ein Datum, das als julianische Tagesnummer gespeichert ist. Die Elementfunktionen lesen, schreiben und speichern Datumseinträge und konvertieren Sie in Datumseinträge des gregorianischen Kalenders.

Typdefinitionen

HowToPrint

```
enum HowToPrint{ Normal, Terse, Numbers, EuropeanNumbers, European };
```

Liste verschiedener Ausgabeformate

DayTy

```
typedef unsigned DayTy;
```

Typ für Tag-Werte.

Month

```
typedef unsigned MonthTy;
```
Typ für Monats-Werte.

YearTy

```
typedef unsigned YearTy;
```
Typ für Jahres-Werte.

JulTy

```
typedef unsigned JulTy;
```
Typ für julianischen Kalender.

Public-Konstruktoren

```
TDate();
```
Erzeugt ein *TDate*-Objekt mit dem aktuellen Datum.

```
TDate( DayTy day, YearTy year );
```
Erzeugt ein *TDate*-Objekt mit dem angegebenem Tag *day* und Jahr *year*. Das Basisdatum für diese Berechnung ist der 31. Dezember des vorigen Jahres. Für *year* == 0 und *day* == 0 wird das Datum 1. Januar 1901 als "Tag null" erzeugt. Beispiel: *TDate(-1,0)* = 31. Dezember 1900 und *TDate(1,0)* = 2. Januar 1901

```
TDate( DayTy day, const char* month, YearTy year);
TDate( DayTy day, MonthTy month, YearTy year);
```
Erzeugt ein *TDate*-Objekt mit den angegebenen Parametern *day*, *month* und *year*.

```
TDate( istream& is );
```
Erzeugt ein *TDate*-Objekt, wobei das Datum aus dem Eingabe-Stream *is* gelesen wird.

```
TDate( const TTime& time );
```
Erzeugt ein *TDate*-Objekt aus dem *TTime*-Objekt *time*.

Public-Elementfunktionen

AsString

```
string AsString() const;
```
Konvertiert das *TDate*-Objekt in ein *string*-Objekt.

Between

```
int Between( const TDate& d1, const TDate& d2 ) const;
```
Liefert 1 zurück, falls das *TDate*-Objekt zwischen *d1* und *d2* liegt.

CompareTo

```
int CompareTo( const TDate & ) const;
```
Liefert 1 zurück, wenn das *TDate*-Objekt größer als der übergebene *TDate*-Parameter ist. Ist das *TDate*-Objekt kleiner als der Parameter, so wird -1 zurückgeliefert; sind die Datumswerte gleich, so wird 0 zurückgeliefert.

Day

```
DayTy Day() const;
```
Liefert den Tag des Jahres (1 - 365) zurück.

DayName const char *DayName(DayTy weekDayNumber);

Liefert den Namen des Wochentags als String, wobei Montag der 1 und Sonntag der 7 entspricht.

DayOfMonth `DayTy DayOfMonth() const;`
Liefert den Monatstag (1 - 31) zurück.

DayOfWeek `DayTy DayOfWeek( const char* dayName );`
Liefert die Tagesnummer zurück, die zum übergebenen String gehört. Der Wert 1 entspricht Montag, der Wert 7 entspricht Sonntag.

DaysInYear `DayTy DaysInYear( YearTy );`
Liefert die Anzahl der Tage im angegebenen Jahr (365 oder 366).

DayWithinMonth `int DayWithinMonth( MonthTy, DayTy, YearTy );`
Liefert 1 zurück, falls der angegebene Tag innerhalb des angegebenen Monats des angegebenen Jahres ist.

FirstDayOfMonth
```
DayTy FirstDayOfMonth() const;
DayTy FirstDayOfMonth( MonthTy month ) const;
```
Liefert die Nummer des ersten Tages des angegebenen Monats. Wenn *month* nicht im Bereich von 1 bis 12 liegt, dann wird der Wert 0 zurückgeliefert.

Hash `unsigned Hash() const;`
Liefert den Hash-Wert von *TDate*.

IndexOfMonth `MonthTy IndexOfMonth( const char *monthName );`
Liefert die Monatsnummer (1 - 12) des Monats *monthname*.

IsValid `int IsValid() const;`
Liefert den Wert 1 zurück, wenn das *TDate*-Objekt ein gültiges Datum enthält. Andernfalls wird 0 zurückgeliefert.

JDay `JulTy Jday( MonthTy, DayTy, YearTy );`
Konvertiert das gegebene gregorianische Datum in die entsprechende julianische Tagesnummer. Der gregorianische Kalender beginnt mit dem 14. September 1752. Diese Funktion ist für ältere Datumswerte nicht gültig. Für ungültige Datumswerte wird 0 zurückgeliefert.

Leap `int Leap() const;`
Liefert 1 zurück, wenn das *TDate*-Jahr ein Schaltjahr ist, ansonsten wird 0 zurückgeliefert.

Max `TDate Max( const TDate& dt ) const;`
Vergleicht das Objektdatum mit dem übergebenen Datum *dt* und liefert das Datum mit der größeren julianischen Tagesnummer zurück.

Min `TDate Min( const TDate& dt ) const;`
Vergleicht das Objektdatum mit dem übergebenen Datum *dt* und liefert das Datum mit der kleineren julianischen Tagesnummer zurück.

Month

`MonthTy Month() const;`

Liefert die Monatsnummer für das *TDate*-Objekt zurück.

Monthname

`const char *MonthName( MonthTy monthNumber );`

Liefert einen String mit dem Monatsnamen für die übergebene Monatsnummer *monthNumber* zurück. Bei ungültigen Monatsnummern wird 0 zurückgeliefert.

NameOfDay

`const char *NameOfDay() const;`

Liefert den Tagesnamen des *TDate*-Objekts zurück.

NameOfMonth

`const char *NameOfMonth() const;`

Liefert den Monatsnamen des *TDate*-Objekts zurück.

Previous

`TDate Previous( const char *dayName ) const;`

Liefert *TDate* des Tags vor *dayName* zurück.

`TDate Previous( DayTy day ) const;`

Liefert *TDate* des Tags vor *day* zurück.

SetPrintOption

`HowToPrint SetPrintOption( HowToPrint h );`

Legt das Ausgabeformat für alle *TDate*-Objekte fest und liefert die alte Einstellung zurück. Siehe *HowToPrint*.

WeekDay

`DayTy WeekDay() const;`

Liefert 1 (Montag) bis 7 (Sonntag) zurück.

Year

`YearTy Year() const;`

Liefert das Jahr des *TDate*-Objekts zurück.

Protected-Elementfunktionen

AssertIndexOfMonth

`static int AssertIndexOfMonth( MonthTy m );`

Liefert 1 zurück, falls *m* zwischen 1 und 12 liegt, ansonsten wird 0 zurückgeliefert.

AssertWeekDayNumber

`static int AssertWeekDayNumber( DayTy d );`

Liefert 1 zurück, falls *d* zwischen 1 und 7 liegt, ansonsten wird 0 zurückgeliefert.

Operatoren

operator <

`int operator < ( const TDate& date ) const;`

Liefert 1 zurück, falls das *TDate*-Objekt zeitlich vor *date* liegt. Andernfalls wird 0 zurückgeliefert.

operator <=

```
int operator <= ( const TDate& date ) const;
```

Liefert 1 zurück, falls das *TDate*-Objekt kleiner oder gleich *date* ist. Andernfalls wird 0 zurückgeliefert.

operator >

```
int operator > ( const TDate& date ) const;
```

Liefert 1 zurück, falls das *TDate*-Objekt größer als *date* ist. Andernfalls wird 0 zurückgeliefert.

operator >=

```
int operator >= ( const TDate& date ) const;
```

Liefert 1 zurück, falls das *TDate*-Objekt größer oder gleich *date* ist. Andernfalls wird 0 zurückgeliefert.

operator ==

```
int operator == ( const TDate& date ) const;
```

Liefert 1 zurück, falls das *TDate*-Objekt gleich *date* ist. Andernfalls wird 0 zurückgeliefert.

operator !=

```
int operator != ( const TDate& date ) const;
```

Liefert 1 zurück, falls das *TDate*-Objekt ungleich *date* ist. Andernfalls wird 0 zurückgeliefert.

operator -

```
JulTy operator - ( const TDate& dt ) const;
```

Subtrahiert *dt* vom *TDate*-Objekt und liefert die Differenz zurück.

operator +

```
friend TDate operator + ( const TDate& dt, int dd );
friend TDate operator + ( int dd, const TDate& dt );
```

Liefert ein neues *TDate*-Objekt zurück, das die Summe aus diesem *TDate*-Objekt und dem Parameter *dd* enthält.

operator -

```
friend TDate operator - ( const TDate& dt, int dd );
```

Subtrahiert *dd* vom *TDate*-Objekt und liefert die Differenz zurück.

operator ++

```
void operator ++ ();
```

Inkrementiert das *TDate*-Objekt um 1.

operator --

```
void operator -- ();
```

Dekrementiert das *TDate*-Objekt um 1.

operator +=

```
void operator += ( int dd );
```

Addiert *dd* zum *TDate*-Objekt.

operator -=

```
void operator -= ( int dd );
```

Subtrahiert *dd* vom *TDate*-Objekt.

operator <<

```
friend ostream& operator << ( ostream& os, const TDate& date );
```

Fügt *date* in den Ausgabe-Stream *os* ein.

operator >>

```
friend istream& operator >> ( istream& is, TDate& date );
```

Liest *date* aus dem Eingabe-Stream *is*.

Struktur TFileStatus file.h

```
struct TFileStatus
{
   TTime createTime;
   TTime modifyTime;
   TTime accessTime;
   long size;
   uint8 attribute;
   char fullName[_MAX_PATH];
};
```

Beschreibt eine Datei durch Erzeugungs-, Änderungs- und Zugriffszeiten, Dateigröße, Attribute und Namen. Siehe auch *TTime*.

Klasse TFile file.h

```
class TFile
```

Die Klasse *TFile* kapselt die Standardeigenschaften und -operationen von Dateien.

Public-Datenelemente

FileNull

```
enum { FileNull };
```

Repräsentiert das Datei-Handle NULL.

Datei-Flags

```
enum{
   ReadOnly    = O_RDONLY,
   ReadWrite   = O_RDWR,
   WriteOnly   = O_WRONLY,
   Create      = O_CREAT | O_TRUNC,
   CreateExcl  = O_CREAT | O_EXCL,
   Append      = O_APPEND,
#if defined( __FLAT__ )
   Compat      = SH_COMPAT,
   DenyNone    = SH_DENYNONE,
#else
   DenyRead    = SH_DENYRD,
   DenyWrite   = SH_DENYWR,
#endif
   DenyRdWr    = SH_DENYRW,
   NoInherit   = O_NOINHERIT
   };
```

Enthält die Dateiübersetzungsmodi und Konstanten zum gemeinsamen Dateizugriff. Siehe Funktionen *open* und *sopen* in Kapitel 3.

```
enum{
   PermRead    = S_IREAD,
   PermWrite   = S_IWRITE,
   PermRdWr    = S_IREAD | S_IWRITE
   };
```
Repräsentiert die Schreib-/Lesezugriffrechte für Dateien. Siehe Funktion *creat* in Kapitel 3.

```
enum{
   Normal      = 0x00,
   RdOnly      = 0x01,
   Hidden      = 0x02,
   System      = 0x04,
   Volume      = 0x08,
   Directory   = 0x10,
   Archive     = 0x20
  };
```
Enthält die Dateitypen.

```
enum seek_dir {
   beg = 0,
   cur = 1,
   end = 2
  };
```
Enthält die Richtungsangaben für Positionierungen des Dateizeigers.

Konstruktoren

```
TFile();
```
Erzeugt ein *TFile*-Objekt mit dem Datei-Handle *FileNull*.

```
TFile( int handle );
```
Erzeugt ein *TFile*-Objekt mit dem Datei-Handle *handle*.

```
TFile( const TFile& file );
```
Erzeugt ein *TFile*-Objekt mit demselben Datei-Handle wie *file*.

```
TFile( const char* name, uint16 access=ReadOnly,
uint16 permission=PermRdWr );
```
Erzeugt ein *TFile*-Objekt und öffnet die Datei *name* mit den angegebenen Attributen. Existiert die Datei nicht, so wird sie erzeugt.

Public-Elementfunktionen

Close

```
int Close();
```
Schließt die Datei. Bei erfolgreicher Ausführung wird ein Wert ungleich 0, ansonsten wird 0 zurückgeliefert.

Flush

`void Flush();`

Alle noch ausstehenden Ein-/Ausgabeoperationen werden durchgeführt.

GetHandle

`int GetHandle() const;`

Liefert das Datei-Handle zurück.

GetStatus

`int GetStatus( TFileStatus& status ) const;`

Setzt den Parameter *status* auf den aktuellen Dateistatus. Bei erfolgreicher Ausführung wird ein Wert ungleich 0, ansonsten wird 0 zurückgeliefert.

`int GetStatus( const char *name, TFileStatus& status );`

Setzt den Parameter *status* auf den Dateistatus der Datei *name*. Bei erfolgreicher Ausführung wird ein Wert ungleich 0, ansonsten wird 0 zurückgeliefert.

IsOpen

`int IsOpen() const;`

Liefert bei geöffneter Datei 1, ansonsten 0 zurück.

Length

`long Length() const;`

Liefert die Dateigröße zurück.

`void Length( long newLen );`

Ändert die Größe der Datei auf *newLen*.

LockRange

`void LockRange( long position, uint32 count );`

Sperrt, beginnend mit *position, count* Bytes der zugehörigen Datei. Siehe auch *UnlockRange*.

Open

`int Open( const char* name, uint16 access, uint16 permission );`

Öffnet die Datei *name* mit den angegebenen Attributen. Die Datei wird dabei erzeugt, wenn sie noch nicht existiert. Bei erfolgreicher Ausführung wird 1, ansonsten wird 0 zurückgeliefert.

Position

`long Position() const;`

Liefert die aktuelle Position des Dateizeigers. Der Rückgabewert -1 zeigt einen Fehler an.

Read

`int Read( void *buffer, int numBytes );`

Liest *numBytes* aus der Datei in den Puffers *buffer*.

`long Read( void huge *buffer, long numBytes );`

Liest *numBytes* aus der Datei in den Puffer *buffer* (Version für 32-Bit Windows).

Remove

`static void Remove( const char *name );`

Löscht die Datei *name*. Bei erfolgreicher Ausführung wird 0, ansonsten -1 zurückgeliefert.

Rename

`static void Rename( const char *oldName, const char *newName );`

Die Datei *oldName* wird in *newName* umbenannt.

Seek `long Seek( long offset, int origin = beg );`
Positioniert den Dateizeiger *offset* Bytes entfernt von *origin*.

SeekToBegin `long SeekToBegin();`
Positioniert den Dateizeiger auf den Dateianfang.

SeekToEnd `long SeekToEnd();`
Positioniert den Dateizeiger auf das Dateiende.

SetStatus `static int SetStatus( const char *name, const TFileStatus& status );`
Setzt den Status der Datei *name* auf *status*.

UnlockRange `void UnlockRange(long Position, uint32 count );`
Entsperrt den Bereich an der angegebenen *Position*. Siehe auch *LockRange*.

Write `int Write( const void *buffer, int numBytes );`
Schreibt *numBytes* Bytes des Puffers *buffer* in die Datei.

`long Write( const void huge *buffer, long numBytes );`
Schreibt *numBytes* Bytes des Puffers *buffer* in die Datei (Version für 32-Bit Windows).

Klasse string — cstring.h

`class string`
Diese Klasse verwendet eine Technik namens »Kopie beim Schreiben«. Mehrere Instanzen eines Strings können dabei mit denselben Daten arbeiten, solange nur lesend auf die Stringdaten zugegriffen wird. Sobald aber ein String seine Daten ändert, wird automatisch eine Kopie angelegt, sofern diese Daten von mehr als einem String referenziert werden.

Typdefitionen

`enum StripType { Leading, Trailing, Both };`
Aufzählungstyp für das Entfernen von Zeichen. Siehe Funktion *strip*.

Konstruktoren

`string();`
Der Standardkonstruktor erzeugt einen String der Länge 0.

`string(const string _FAR &s);`
Der Kopierkonstruktor erzeugt einen String, der eine Kopie des Strings *s* enthält.

`string( const string _FAR &s, size_t n );`
Ein String mit einer Kopie von *n* Bytes des Strings *s* wird erzeugt.

`string(const char _FAR *cp);`
Es wird ein String erzeugt, in den die Bytes, auf die *cp* zeigt, bis zum Auftreten der ersten 0 kopiert werden (Konvertierung von *char**).

`string(const char _FAR *cp, size_t n);`
Es wird ein String erzeugt, in den *n* Bytes, beginnend mit der durch *cp* angegebenen Position, kopiert werden.

`string( char c );`
Es wird ein String erzeugt, der das Zeichen *c* enthält.

`string( char c, size_t n );`
Es wird ein String erzeugt, der das Zeichen *c n*-mal enthält.

`string( signed char c );`
Es wird ein String erzeugt, der das Zeichen *c* enthält.

`string( signed char c, size_t n );`
Es wird ein String erzeugt, der das Zeichen *c n*-mal enthält.

`string( unsigned char c );`
Es wird ein String erzeugt, der das Zeichen *c* enthält.

`string( unsigned char c, size_t n );`
Es wird ein String erzeugt, der das Zeichen *c n*-mal enthält.

`string(const TSubString _FAR &ss);`
Es wird ein String aus dem Teilstring *ss* erzeugt.

```
string( const char __far *cp );
string( const char __far *cp, size_t n );
```
Es werden Strings für die Windows-Speichermodelle Small und Medium erzeugt.

`string( HINSTANCE instance, UINT id, int len = 255 )`
Windows-Version zur Erzeugung eines Strings aus einer Ressource.

Destruktor

`~String();`
Entfernt den String und alle vom *string*-Objekt reservierten Ressourcen.

Public-Elementfunktionen

ansi_to_oem

`void ansi_to_oem();`
Konvertiert den String vom ANSI-Zeichensatz in den OEM-Zeichensatz (nur für Windows).

append

`string _FAR & append( const string _FAR &s )`
Hängt den String *s* an das *string*-Objekt an.

`string _FAR & append( const string _FAR &s, size_t n );`
Hängt die ersten *n* Zeichen des String *s* an das *string*-Objekt an.

`string _FAR & append( const char _FAR *cp, size_t n );`
Hängt die ersten *n* Zeichen des Zeichen-Array *cp* an das *string*-Objekt an.

assign

`string _FAR & assign( const string _FAR &s );`
Weist den String *s* dem *string*-Objekt zu. Siehe auch *operator* =.

`string _FAR & assign( const string FAR &s, size_t n );`
Weist *n* Zeichen des Strings *s* dem *string*-Objekt zu. Siehe auch *operator* =.

compare

`int compare(const string _FAR &s);`
Vergleicht das *string*-Objekt mit dem String *s*. *compare* liefert Integerwerte kleiner, gleich oder größer 0, abhängig davon, ob das *string*-Objekt kleiner, gleich oder größer *s* ist.

`int compare(const string _FAR &s, size_t n );`
Vergleicht nicht mehr als die ersten *n* Zeichen des *string*-Objekts mit *s*.

contains

`int contains(const char _FAR * pat) const;`
Liefert 1 zurück, wenn *pat* im *string*-Objekt enthalten ist. Ansonsten wird der Wert 0 zurückgeliefert.

`int contains(const string _FAR & s) const`
Liefert 1 zurück, wenn der String *s* im *string*-Objekt enthalten ist. Ansonsten wird 0 zurückgeliefert.

copy

`size_t copy( char _FAR *cb, size_t n )`
Kopiert höchstens *n* Zeichen des *string*-Objekts in das durch *cp* angegebene Zeichen-Array. *copy* liefert die Anzahl der kopierten Zeichen zurück.

`size_t copy( char _FAR *cb, size_t n, size_t pos );`
Kopiert, beginnend mit der Position *pos*, höchstens *n* Zeichen des *string*-Objekts in das durch *cp* angegebene Zeichen-Array. *copy* liefert die Anzahl der kopierten Zeichen zurück.

`string copy() const throw( xalloc );`
Liefert eine Kopie des *Strings* zurück.

c_str

`const char _FAR *c_str() const;`
Liefert einen Zeiger auf ein nullterminiertes Zeichen-Array, das dieselben Zeichen enthält wie das *string*-Objekt. Der zurückgelieferte Zeiger kann dabei auf den tatsächlichen Stringinhalt oder eine beim Funktionsaufruf angelegte Kopie davon zeigen. Die Auswirkungen einer Veränderung des Array-Inhalts sind ebenso undefiniert wie die Ergebnisse eines Array-Zugriffs nach der Ausführung einer nicht-**const** Elementfunktion.

Konvertierungen eines *string*-Objekts nach char* sind gefährlich, da sie die Klassengrenzen verletzen und zu undefinierten Zeigern führen können. Daher verfügt die Klasse *string* nicht über eine implizite Typkonver-

tierung nach char* sondern stellt für diesen Zweck die Funktion *c_str* bereit.

find

`size_t find( const string _FAR &s );`
Sucht den Strings *s* im *string*-Objekt. Wird *s* gefunden, so liefert die Funktion die Anfangsposition von *s* im *string*-Objekt zurück. Wird *s* nicht gefunden, so wird NPOS zurückgeliefert.

`size_t find( const string _FAR &s, size_t pos );`
Sucht den Strings *s* im *string*-Objekt ab der Position *pos*. Wird *s* gefunden, so liefert die Funktion die Anfangsposition von *s* im *string*-Objekt zurück. Wird *s* nicht gefunden, so wird NPOS zurückgeliefert.

`size_t find( const TRegexp _FAR &pat, size_t i = 0 );`
Durchsucht ab Position *i* den String nach Mustern, die dem regulären Ausdruck *pat* entsprechen. Zurückgeliefert wird die Position von *pat* innerhalb des *string*-Objekts. Wird *pat* nicht gefunden, so wird NPOS zurückgeliefert.

`size_t find( const TRegexp _FAR &pat, size_t _FAR *ext, size_t i = 0 ) const;`
Durchsucht ab Position *i* den String nach Mustern, die dem regulären Ausdruck *pat* entsprechen. Der Parameter *ext* enthält die Länge des passenden Teilstrings, sofern dieser gefunden wurde. Zurückgeliefert wird die Position von *pat* innerhalb des *string*-Objekts. Wird *pat* nicht gefunden, so wird NPOS zurückgeliefert.

find_first_of

`size_t find_first_of( const string _FAR &s ) const;`
Sucht die erste Position im *string*-Objekt, in der ein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche liefert *find_first_of* die Position des Zeichens. Wenn die Suche fehlschlägt oder wenn `pos > length()` gilt, dann wird 0 zurückgeliefert.

`size_t find_first_of( const string _FAR &s, size_t pos ) const;`
Sucht die erste Position im *string*-Objekt, in der ein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche wird *pos* auf die Position des Zeichens innerhalb des *string*-Objekts gesetzt und *find_first_of* liefert 1 zurück. Wenn die Suche fehlschlägt oder wenn `pos > length()` gilt, dann wird 0 zurückgeliefert.

find_first_not_of

`size_t find_first_not_of( const string _FAR &s ) const;`
Sucht die erste Position im *string*-Objekt, in der kein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche liefert *find_first_not_of* die Position des Zeichens. Wenn die Suche fehlschlägt oder wenn `pos > length()` gilt, dann wird 0 zurückgeliefert.

`size_t find_first_not_of( const string _FAR &s, size_t pos ) const;`
Sucht die erste Position im *string*-Objekt, in der ein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche wird *pos* auf die Position des

Zeichens innerhalb des *string*-Objekts gesetzt und *find_first_not_of* liefert 1 zurück. Wenn die Suche fehlschlägt oder wenn `pos > length()` gilt, dann wird 0 zurückgeliefert.

find_last_of

`size_t find_last_of( const string _FAR &s ) const;`
Sucht die letzte Position im *string*-Objekt, in der ein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche liefert *find_last_of* die Position des Zeichens. Wenn die Suche fehlschlägt oder wenn `pos > length()` gilt, dann wird 0 zurückgeliefert.

`size_t find_last_of( const string _FAR &s, size_t pos ) const;`
Sucht die letzte Position im *string*-Objekt, in der ein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche wird *pos* auf die Position des Zeichens innerhalb des *string*-Objekts gesetzt und *find_last_of* liefert 1 zurück. Wenn die Suche fehlschlägt oder wenn `pos > length()` gilt, dann wird 0 zurückgeliefert.

find_last_not_of

`size_t find_first_not_of( const string _FAR &s ) const;`
Sucht die letzte Position im *string*-Objekt, in der kein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche liefert *find_last_not_of* die Position des Zeichens. Wenn die Suche fehlschlägt oder wenn `pos > length()`aus gilt, dann wird 0 zurückgeliefert.

`size_t find_last_not_of( const string _FAR &s, size_t pos ) const;`
Sucht die letzte Position im *string*-Objekt, in der kein Zeichen aus dem String *s* auftritt. Bei erfolgreicher Suche wird *pos* auf die Position des Zeichens innerhalb des *string*-Objekts gesetzt und *find_last_not_of* liefert 1 zurück. Wenn die Suche fehlschlägt oder wenn `pos > length()` gilt, dann wird 0 zurückgeliefert.

get_at

`char get_at( size_t pos ) const throw( outofrange );`
Liefert das Zeichen an der angegebenen Position zurück. Wenn `pos > length()-1` wird eine *outofrange* Exception ausgelöst.

get_case_sensitive_flag

`static int get_case_sensitiveFlag();`
Liefert 0 zurück, falls bei Stringvergleichen Groß-/Kleinschreibung berücksichtigt wird. Andernfalls wird 1 zurückgeliefert.

get_initial_capacity

`static unsigned get_initial_capacity();`
Liefert die Anzahl der Zeichen, die ohne Größenänderung in den String passen.

get_max_waste

`static unsigned get_max_waste();`
Liefert die Größe des verfügbaren freien Stringbereichs zurück, nachdem die Stringgröße geändert wurde.

get_paranoid_check `static int get_paranoid_check();`
Liefert 1 zurück, falls die »paranoid-Überprüfung« eingeschaltet ist, andernfalls wird 0 zurückgeliefert.

get_resize_increment `static unsigned get_resize_increment();`
Liefert den Inkrementwert zurück, der bei Größenänderungen des Strings verwendet wird.

get_skipwhitespace_flag `static int get_skipwhitespace_flag()`
Liefert 1 zurück, wenn Whitespace-Zeichen übersprungen werden. Andernfalls wird 0 zurückgeliefert.

hash `unsigned hash() const;`
Liefert einen Hash-Wert zurück.

initial_capacity `static size_t initial_capacity(size_t ic = 63);`
Legt die Speicherreservierungskapazität des Strings fest.

insert `string _FAR &insert( size_t pos, const string _FAR &s )`
Fügt den String *s* an der Position *pos* in das *string*-Objekt ein. *insert* liefert eine Referenz auf den Ergebnisstring zurück.

`string _FAR &insert( size_t pos, const string _FAR &s, size_t n );`
Fügt *n* Zeichen des Strings *s* an der Position *pos* in das *string*-Objekt ein. *insert* liefert eine Referenz auf den Ergebnisstring zurück.

is_null `int is_null() const;`
Liefert bei leerem String 1, ansonsten 0 zurück.

length `unsigned length() const;`
Liefert die Zeichenanzahl im *string*-Objekt zurück. Da in einem String Nullzeichen gespeichert werden können, kann das Ergebnis von *length()* größer sein als *strlen(c_str())*.

max_waste `static size_t MaxWaste(size_t mw = 63);`
Legt den maximalen freien Stringbereich fest. Sobald ein Bereich dieser Größe frei ist, wird die Stringgröße geändert.

oem_to_ansi `void oem_to_ansi();`
Funktion für die Konvertierung des *string*-Objekts vom OEM in den ANSI-Zeichensatz (nur für Windows).

prepend `string _FAR &prepend( const string _FAR &s )`
Fügt den String *s* am Anfang des *string*-Objekts ein.

`string _FAR &prepend( const string _FAR &s, size_t n );`
Fügt die ersten *n* Zeichen des Strings *s* am Anfang des *string*-Objekts ein.

`string _FAR &prepend( const char _FAR *cp );`
Fügt den Zeichen-Array *cp* am Anfang des *string*-Objekts ein.

`string _FAR &prepend( const char _FAR *cp, size_t n );`
Fügt die ersten *n* Zeichen des Array *cp* am Anfang des *string*-Objekts ein.

put_at

`void put_at( size_t pos, char c ) throw( outofrange );`
Ersetzt das Zeichen an Position *pos* durch *c*. Wenn `pos == length` gilt, dann hängt *put_at* das Zeichen *c* an das *string*-Objekt an. Bei `pos > length` wird eine *outofrange* Exception ausgelöst.

read_file

`istream _FAR &read_file(istream _FAR &is);`
Aus dem Eingabe-Stream *is* wird gelesen bis EOF oder ein Nullzeichen erreicht wird.

read_line

`istream _FAR &read_line(istream _FAR &is);`
Aus dem Eingabe-Stream *is* wird gelesen bis EOF oder ein Zeilenvorschub erreicht wird.

read_string

`istream _FAR &read_string(istream _FAR &is);`
Aus dem Eingabe-Stream *is* wird gelesen bis EOF oder ein Nullzeichen erreicht wird.

read_to_delim

`istream _FAR &read_to_delim(istream _FAR &is, char delim='\n');`
Aus dem Eingabe-Stream *is* wird gelesen bis EOF oder das Zeichen *delim* erreicht wird.

read_token

`istream _FAR &read_token(istream _FAR &is);`
Aus dem Eingabe-Stream *is* wird gelesen bis ein Whitespace-Zeichen erreicht wird. Beachten Sie, daß diese Funktion alle führenden Whitespace-Zeichen überspringt.

rfind

`size_t rfind( const string _FAR &s );`
Sucht das letzte Vorkommen des Strings *s* im *string*-Objekt. Wird der String gefunden, so liefert die Funktion die Anfangsposition von *s* innerhalb des *string*-Objekts zurück. Andernfalls wird NPOS zurückgeliefert.

`size_t rfind( const string _FAR &s, size_t pos );`
Sucht das letzte Vorkommen des Strings *s* im *string*-Objekt bis höchsten zur Position *pos*. Wird der String gefunden, so liefert die Funktion die Anfangsposition von *s* innerhalb des *string*-Objekts zurück. Andernfalls wird NPOS zurückgeliefert. Siehe auch *find*.

remove

`string _FAR &remove( size_t pos );`
Entfernt beginnend mit Position *pos* alle folgenden Zeichen des *string*-Objekts und liefert eine Referenz auf den resultierenden String zurück.

`string _FAR &remove( size_t pos, size_t n );`
Entfernt beginnend mit Position *pos* höchstens *n* Zeichen des *string*-Objekts und liefert eine Referenz auf den resultierenden String zurück.

replace

`string _FAR &replace( size_t pos, size_t n, const string _FAR &s );`

Entfernt beginnend mit Position *pos* höchstens *n* Zeichen des *string*-Objekts und ersetzt diese mit einer Kopie des Strings *s*. *replace* liefert eine Referenz auf den resultierenden String zurück.

`string _FAR &replace( size_t pos, size_t n1, const string _FAR &s, size_t n2 );`

Entfernt beginnend mit Position *pos* höchstens *n1* Zeichen aus dem *string*-Objekt und ersetzt sie durch die ersten *n2* Zeichen des Strings *s*. *replace* liefert eine Referenz auf den resultierenden String zurück.

reserve

`size_t reserve() const;`

Liefert einen implementationsabhängigen Wert zurück, der die aktuelle interne Speichergröße kennzeichnet. Der zurückgegebene Wert ist stets größer oder gleich `length()`.

`void reserve( size_t ic );`

Hinweis für die Implementierung, daß das *string*-Objekt letztendlich *ic* Bytes Speicherplatz benötigen könnte.

resize

`void resize(size_t m);`

Ändert die Stringgröße auf *m* Zeichen. Leerzeichen werden dabei nötigenfalls hinzugefügt bzw. entfernt.

resize_increment

`static size_t resize_increment(size_t ri = 64);`

Legt den Inkrementwert für automatische Größenänderungen fest.

set_case_sensitive

`static int set_case_sensitive(int tf = 1);`

Legt die Berücksichtigung von Groß-/Kleinschreibung fest. 1 bedeutet, daß Groß-/Kleinschreibung berücksichtigt wird. 0 bedeutet, daß sie nicht berücksichtigt wird.

set_paranoid_check

`static int set_paranoid_check(int ck = 1);`

Bei der Suche nach Strings wird eine Hash-Tabelle verwendet. Dabei ist es möglich, daß mehr als ein String denselben Hash-Wert besitzt. Ein Aufruf von *set_paranoid_check* mit *ck* gleich 1 gewährleistet, daß der gefundene und der gesuchte String mit Hilfe der Funktion *strcmp* verglichen werden. Ein Aufruf von *set_paranoid_check* mit *ck* gleich 0 schaltet diese abschließende Überprüfung aus.

skip_whitespace

`static int skip_whitespace(int sk = 1);`

Mit *sk* gleich 1 werden Whitespace-Zeichen beim Lesen von Tokens übersprungen, bei *sk* gleich 0 werden sie nicht übersprungen.

strip

`TSubString strip( StripType s = Trailing, char c=' ');`

Entfernt das Zeichen *c* vom Anfang, vom Ende oder von Anfang und Ende des Strings – je nachdem, welchen Wert *StripType* hat.

substr

`string substr( size_t pos ) const`

Erzeugt einen String mit einer Kopie der Zeichen des *string*-Objekts von der Position *pos* bis zum Ende.

`string substr( size_t pos, size_t n ) const`
Erzeugt einen String mit einer Kopie von höchstens *n* Zeichen des *string*-Objekts beginnend mit der Position *pos*.

substring

`TSubString substring( const char _FAR *cp );`
Erzeugt ein *TSubString*-Objekt mit einer Kopie des durch *cp* angegebenen Zeichen-Array.

`const TSubString substring( const char _FAR *cp );`
Erzeugt ein *TSubString*-Objekt mit einer Kopie des durch *cp* angegebenen Zeichen-Array.

`TSubString substring( const char _FAR *cp, size_t start );`
Erzeugt ein *TSubString*-Objekt mit einer Kopie des durch *cp* angegebenen Zeichen-Array, beginnend mit dem Zeichen an Position *start*.

`const TSubString substring( const char _FAR *cp, size_t start ) const`
Erzeugt ein *TSubString*-Objekt mit einer Kopie des durch *cp* angegebenen Zeichen-Array, beginnend mit dem Zeichen an Position *start*.

to_lower

`void to_lower();`
Ersetzt alle Großbuchstaben des *string*-Objekts durch Kleinbuchstaben.

to_upper

`void to_upper();`
Ersetzt alle Kleinbuchstaben des *string*-Objekts durch Großbuchstaben.

Protected-Elementfunktionen

assert_element

`void assert_element( size_t pos ) const;`
Löst eine *outofrange* Exception aus, wenn ein ungültiges Element angegeben wird.

assert_index

`void assert_index( size_t pos ) const;`
Löst eine *outofrange* Exception aus, wenn ein ungültiger Indexwert angegeben wird.

cow

`void cow();`
Kopie bei Schreiboperationen (copy on write). Mehrere Instanzen eines Strings können solange dieselben Daten referenzieren, wie nur lesend auf diese zugegriffen wird. Sobald ein Schreibzugriff stattfindet, wird *cow* aufgerufen und eine Kopie der Daten angelegt, sofern diese von mehreren *string*-Objekten referenziert werden.

valid_element

`int valid_element( size_t pos ) const;`
Liefert 1 zurück, wenn *pos* ein Element des Strings ist. Andernfalls wird 0 zurückgeliefert.

valid_index

```
int valid_index( size_t pos ) const;
```
Liefert 1 zurück, wenn *pos* ein gültiger Index des Strings ist. Andernfalls wird 0 zurückgeliefert.

Operatoren

operator =

```
string _FAR & operator=(const string _FAR &s);
```
Wenn das *string*-Objekt mit dem übergebenen *String*-Parameter übereinstimmt, dann bewirkt der Zuweisungsoperator nichts. Andernfalls führt er alle erforderlichen Maßnahmen durch, um die vom *string*-Objekt belegten Ressourcen freizugeben und danach den String *s* in das *string*-Objekt zu kopieren.

operator +=

```
string _FAR & operator += (const String _FAR &s);
```
Hängt den String *s* an das *string*-Objekt an.

```
string _FAR & operator +=(const char _FAR *cp);
```
Hängt den Inhalt von *cp* an das *string*-Objekt an.

operator +

```
friend String _Cdecl _FARFUNC operator+(const String _FAR &, const char
                                         _FAR *cp);
```
Verbindet die beiden Strings *s* und *cp*.

operator []

```
char _FAR & operator[](size_t pos);
```
Liefert eine Referenz auf das Zeichen an Position *pos* zurück.

```
char operator[](size_t pos) const;
```
Liefert das Zeichen an Position *pos*.

operator ()

```
char _FAR & operator()(size_t pos);
```
Liefert eine Referenz auf das Zeichen an Position *pos* zurück.

```
TSubString operator()(size_t start, size_t len);
```
Liefert den Teilstring der Länge *len* zurück, der bei Position *start* beginnt.

```
TSubString operator()(const TRegexp _FAR & re);
```
Liefert das erste Auftreten eines Teilstrings zurück, der dem regulären Ausdruck *re* entspricht.

```
TSubString operator()(const TRegexp _FAR & re, size_t start);
```
Liefert das erste Auftreten eines Teilstrings zurück, der dem regulären Ausdruck *re* entspricht. Gesucht wird ab der Position *start*.

```
char operator()(size_t pos) const;
```
Liefert das Zeichen an Position *pos* zurück.

```
const TSubString operator()(size_t start, size_t len) const;
```
Liefert den Teilstring der Länge *len* zurück, der bei Position *start* beginnt.

```
const TSubString operator()(const TRegexp _FAR & pat) const;
```

Liefert das erste Auftreten eines Teilstrings zurück, der dem regulären Ausdruck *re* entspricht.

```
const TSubString operator()(const TRegexp _FAR & pat, size_t start) const;
```

Liefert das erste Auftreten eines Teilstrings zurück, der dem regulären Ausdruck *re* entspricht. Gesucht wird ab der Position *start*.

operator ==

```
friend int operator == ( const String _FAR &s1, const String _FAR &s2 );
```

Testet die Gleichheit der Strings *s1* und *s2*. Zwei Strings sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator == liefert 1 zurück, wenn die Strings gleich sind. Sind die Strings nicht gleich, so wird der Wert 0 zurückgeliefert.

```
friend int operator == ( const String _FAR &s, const char _FAR *cp );
friend int operator == ( const char _FAR *cp, const String _FAR &s );
```

Testet die Gleichheit zwischen dem String *s* und dem durch *cp* angegebenen Zeichen-Array. Beide sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator == liefert 1 zurück, wenn die Strings gleich sind. Sind die Strings nicht gleich, so wird der Wert 0 zurückgeliefert.

operator !=

```
friend int operator != ( const String _FAR &s1, const String _FAR &s2 );
```

Testet die Ungleichheit der Strings *s1* und *s2*. Zwei Strings sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator != liefert 1 zurück, wenn die Strings ungleich sind. Sind die Strings gleich, so wird der Wert 0 zurückgeliefert.

```
friend int operator != ( const String _FAR &s, const char _FAR *cp );
friend int operator != ( const char _FAR *cp, const String _FAR &s );
```

Testet die Ungleichheit zwischen dem String *s* und dem durch *cp* angegebenen Zeichen-Array. Beide sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator != liefert 1 zurück, wenn die Strings ungleich sind. Sind die Strings gleich, so wird der Wert 0 zurückgeliefert.

operator <

```
friend int operator < ( const String _FAR &s1, const String _FAR &s2 );
```

Vergleicht die beiden Strings *s1* und *s2*. Die Funktion liefert den Wert 1 zurück, falls *s1* kleiner als *s2* ist. Andernfalls wird 0 zurückgeliefert.

```
friend int operator < ( const String _FAR &s, const char _FAR *cp );
friend int operator < ( const char _FAR *cp, const String _FAR &s );
```

Vergleicht den String *s* mit dem durch *cp* angegebenen Zeichen-Array. Die Funktion liefert 1 zurück, falls die linke Seite des Ausdrucks kleiner als die rechte ist. Andernfalls wird 0 zurückgeliefert.

operator <=

```
friend int operator <= ( const String _FAR &s1, const String _FAR &s2 );
```

Vergleicht die beiden Strings *s1* und *s2*. Die Funktion liefert den Wert 1 zurück, falls *s1* kleiner oder gleich *s2* ist. Andernfalls wird 0 zurückgeliefert.

```
friend int operator <= ( const String _FAR &s, const char _FAR *cp );
friend int operator <= ( const char _FAR *cp, const String _FAR &s );
```

Vergleicht den String *s* mit dem durch *cp* angegebenen Zeichen-Array. Die Funktion liefert 1 zurück, falls die linke Seite des Ausdrucks kleiner oder gleich der rechten ist. Andernfalls wird 0 zurückgeliefert.

operator >

```
friend int operator > ( const String _FAR &s1, const String _FAR &s2 );
```

Vergleicht die beiden Strings *s1* und *s2*. Die Funktion liefert den Wert 1 zurück, falls *s1* größer als *s2* ist. Andernfalls wird 0 zurückgeliefert.

```
friend int operator > ( const String _FAR &s, const char _FAR *cp );
friend int operator > ( const char _FAR *cp, const String _FAR &s );
```

Vergleicht den String *s* mit dem durch *cp* angegebenen Zeichen-Array. Die Funktion liefert 1 zurück, falls die linke Seite des Ausdrucks größer als die rechte ist. Andernfalls wird 0 zurückgeliefert.

operator >=

```
friend int operator >= ( const String _FAR &s1, const String _FAR &s2 );
```

Vergleicht die beiden Strings *s1* und *s2*. Die Funktion liefert den Wert 1 zurück, falls *s1* größer oder gleich *s2* ist. Andernfalls wird 0 zurückgeliefert.

```
friend int operator >= ( const String _FAR &s, const char _FAR *cp );
friend int operator >= ( const char _FAR *cp, const String _FAR &s );
```

Vergleicht den String *s* mit dem durch *cp* angegebenen Zeichen-Array. Die Funktion liefert 1 zurück, falls die linke Seite des Ausdrucks größer oder gleich der rechten ist. Andernfalls wird 0 zurückgeliefert.

operator >>

```
friend ipstream _FAR & operator >> ( ipstream _FAR & is, string _FAR &str );
```

Liest den String *str* aus dem Eingabe-Stream *is*.

Verwandte globale Funktionen und Operatoren

operator >>

```
istream _FAR & _Cdecl _FARFUNC operator>>(istream _FAR &is, string _FAR &s);
```

Verhält sich genau wie `operator>>(istream&, char *)` (siehe Kapitel 5) und liefert eine Referenz auf *is* zurück.

operator <<

```
ostream _FAR & _Cdecl _FARFUNC operator<<(ostream _FAR &os, const string
                                        _FAR & s);
```

Verhält sich genau wie `operator<<(ostream&, char *)` (siehe Kapitel 5) mit dem einzigen Unterschied, daß beim Auftreten eines Nullzeichens kein Abbruch erfolgt. Liefert eine Referenz auf *os* zurück.

```
opstream _FAR& _Cdecl operator << ( opstream _FAR & os, const string
                                    _FAR & str );
```

Fügt den String *str* in den persistenten Ausgabe-Stream *os* ein.

operator +

```
string _Cdecl _FARFUNC operator + ( const char _FAR *cp, const string
                                    _FAR & s);
```
Verknüpft **cp* mit dem String *s*.

```
string _Cdecl _FARFUNC operator + ( const string _FAR &s1, const string
                                    _FAR &s2 );
```
Verknüpft die Strings *s1* und *s2*.

getline

```
istream _FAR & _Cdecl getline( istream _FAR &is, string _FAR &s );
```
Verhält sich wie `istream::getline(chptr, NPOS)` mit der Ausnahme, daß nicht in ein Zeichen-Array sondern in ein *string*-Objekt geschrieben wird. *getline* liefert eine Referenz auf *is* zurück.

```
istream _FAR & _Cdecl getline( istream _FAR &is, string _FAR &s, char c );
```
Verhält sich wie `istream::getline(cb, NPOS, c)` mit der Ausnahme, daß nicht in ein Zeichen-Array sondern in ein *string*-Objekt geschrieben wird. *getline* liefert eine Referenz auf *is* zurück.

to_lower

```
String _Cdecl _FARFUNC to_lower( const string _FAR &s );
```
Ersetzt alle Großbuchstaben des *string*-Objekts durch Kleinbuchstaben.

to_upper

```
String _Cdecl _FARFUNC to_upper( const string _FAR &s );
```
Ersetzt alle Kleinbuchstaben des *string*-Objekts durch Großbuchstaben.

Klasse TSubString — cstring.h

```
class TSubString
```
Ermöglicht den Zugriff auf bestimmte Teilstrings.

Public-Elementfunktionen

get_at

```
char get_at( size_t pos ) const;
```
Liefert das Zeichen an der angegebenen Position zurück. Gilt `pos > length()-1`, so wird eine Exception ausgelöst. Siehe auch *put_at*.

is_null

```
int is_null() const;
```
Liefert bei leerem String den Wert 1, ansonsten den Wert 0 zurück.

length

```
size_t length() const;
```
Liefert die Länge des Teilstrings zurück.

put_at

```
void put_at( size_t pos, char c );
```
Ersetzt das Zeichen an Position *pos* durch *c*. Gilt `pos == length()`, dann wird *c* an den String angehängt. Bei `pos > length()` wird eine Exception ausgelöst.

start

```
int start() const;
```

Liefert den Index des Anfangszeichens zurück.

to_lower

```
void to_lower();
```

Ersetzt alle Großbuchstaben des Teilstrings durch Kleinbuchstaben.

to_upper

```
void to_upper();
```

Ersetzt alle Kleinbuchstaben des Teilstrings durch Großbuchstaben.

Protected-Elementfunktionen

assert_element

```
int assert_element( size_t pos ) const;
```

Liefert den Wert 1 zurück, falls *pos* eine gültige Indexposition im Teilstring ist. Andernfalls wird 0 zurückgeliefert.

Operatoren

operator =

```
TSubString _FAR & operator=( const string _FAR &s );
```

Kopiert *s* in das *string*-Objekt.

operator ==

```
int operator==(const char _FAR * cp) const;
```

Testet die Gleichheit zwischen dem *TSubString*-Objekt und dem durch *cp* angegebenen Zeichen-Array. Zwei Strings sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator == liefert 1 zurück, wenn die Strings gleich sind. Sind die Strings nicht gleich, so wird der Wert 0 zurückgeliefert.

```
int operator==(const string _FAR & s) const;
```

Testet die Gleichheit zwischen dem *TSubString*-Objekt und dem String *s*. Beide sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator == liefert 1 zurück, wenn die Strings gleich sind. Sind die Strings nicht gleich, so wird der Wert 0 zurückgeliefert.

operator !=

```
int operator!=(const char _FAR * cp) const;
```

Testet auf Ungleichheit zwischen dem *TSubString*-Objekt und dem durch *cp* angegebenen Zeichen-Array. Zwei Strings sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator != liefert 1 zurück, wenn die Strings ungleich sind. Sind die Strings gleich, so wird der Wert 0 zurückgeliefert.

```
int operator!=(const string _FAR & s) const;
```

Testet auf Ungleichheit zwischen dem *TSubString*-Objekt und dem String *s*. Zwei Strings sind gleich, wenn sie dieselbe Länge haben und wenn die jeweils korrespondierenden Zeichen gleich sind. Der Operator != liefert 1

zurück, wenn die Strings ungleich sind. Sind die Strings gleich, so wird der Wert 0 zurückgeliefert.

operator ()

`char _FAR & operator()(size_t pos);`
Liefert eine Referenz auf das Zeichen an Position *pos* zurück.

`char operator()(size_t pos) const;`
Liefert das Zeichen an Position *pos* zurück.

operator []

`char _FAR & operator[](size_t pos);`
Liefert eine Referenz auf das Zeichen an Position *pos* zurück.

`char operator[](size_t pos) const;`
Liefert das Zeichen an Position *pos* zurück.

operator !

`int operator!() const;`
Liefert den Wert 1 zurück, wenn der Teilstring nicht leer ist.

Klasse TCriticalSection — thread.h

`class TCriticalSection`
Die Klasse *TCriticalSection* stellt eine systemunabhängige Schnittstelle für kritische Bereiche in Threads zur Verfügung. *TCriticalSection*-Objekte können zusammen mit *TCriticalSection::Lock*-Objekten sicherstellen, daß jeder der geschützten Code-Bereiche zu einem Zeitpunkt nur von jeweils höchstens einem Thread ausgeführt wird. Siehe auch *TCriticalSection::Lock*.

Konstruktoren

`TCriticalSection();`
Erzeugt ein *TCriticalSection*-Objekt.

Destruktor

`~TCriticalSection();`
Entfernt ein *TCriticalSection*-Objekt.

Klasse TCriticalSection::Lock — thread.h

`class TCriticalSection::Lock`
Diese verschachtelte Klasse verwaltet das Sperren und Entsperren von kritischen Bereichen. Hierzu ein Beispiel:

```
TCriticalSection LockF;
void f()
{
   TCriticalSection::Lock(LockF);

   // Verarbeitung im kritischen Bereich
}
```

Nur jeweils ein Thread darf den Code im kritischen Bereich innerhalb der Funktion *f* ausführen.

Konstruktoren

```
Lock( const TCriticalSection & );
```

Fordert die Sperrung des *TCriticalSection*-Objekts an. Sofern kein *Lock*-Objekt eines anderen Thread eine Sperre auf dieses *TCriticalSection*-Objekt gesetzt hat, wird der Anforderung stattgegeben und die Ausführung fortgesetzt. Andernfalls wird der anfragende Thread angehalten bis die Sperre wieder aufgehoben wird.

Destruktor

```
~Lock();
```

Hebt die Sperre wieder auf.

Klasse TMutex thread.h

```
class TMutex
```

Die Klasse *TMutex* stellt eine systemunabhängige Schnittstelle für kritische Bereiche in Threads zur Verfügung. *TMutex*-Objekte können zusammen mit *TMutex::Lock*-Objekten sicherstellen, daß jeder der geschützten Code-Bereiche nur von jeweils höchstens einem Thread ausgeführt wird.

Der Unterschied zwischen den Klassen *TCriticalSection* und *TMutex* besteht darin, daß beim Sperren mit *TMutex* eine Zeitschranke angegeben werden kann und daß ein *TMutex*-Objekt ein HANDLE besitzt, das außerhalb der Klasse benutzt werden kann. Dies entspricht dem Unterschied, den Windows NT zwischen CRITICALSECTION und Mutex macht. Unter Windows NT ist ein *TCriticalSection*-Objekt wesentlich schneller als ein *TMutex*-Objekt. Bei Betriebssystemen, die diese Unterscheidung nicht kennen, können *TCriticalSection* und *TMutex* dieselbe zugrundeliegende Implementierung verwenden. Ein Geschwindigkeitsvorteil wie unter Windows NT tritt dann nicht auf.

Konstruktoren

`TMutex();`
Erzeugt ein *TMutex*-Objekt.

Destruktor

`~TMutex();`
Entfernt ein *TMutex*-Objekt.

Operatoren

`Operator HANDLE() const;`
Liefert das Handle des zugrundeliegenden Windows NT Semaphorenobjekts zurück.

Klasse TMutex::Lock thread.h

Diese geschachtelte Klasse verwaltet das Sperren und Entsperren von *TMutex*-Objekten.

Konstruktoren

`Lock( const TMutex&, unsigned long timeOut = NoLimit );`
Fordert die Sperrung des *TMutex*-Objekts an. Sofern kein *Lock*-Objekt eines anderen Thread dieses *TMutex*-Objekt gesperrt hat, wird der Anforderung stattgegeben und die Ausführung fortgesetzt. Andernfalls wird der anfragende Thread angehalten bis die Sperre wieder aufgehoben wird oder die per *timeout* angegebene Zeit abgelaufen ist.

Public-Elementfunktionen

Release

`void Release();`
Hebt die Sperre des *TMutex*-Objekts wieder auf.

Klasse TSync thread.h

TSync stellt eine systemunabhängige Schnittstelle für die Entwicklung von Klassen zur Verfügung, die sich wie Monitore verhalten – Klassen, in

denen zu einem Zeitpunkt jeweils immer nur eine Elementfunktion ausgeführt werden kann. *TSync* verwendet *TCriticalSection*, hat keine public-Elemente und kann nur als Basisklasse eingesetzt werden. Siehe auch *TSync::Lock*. Für die Verwendung von *TSync* hier ein Beispiel:

```
class ThreadSafe : private TSync
{
public:
   void f();
   void g();
private:
   int i;
};

void ThreadSafe::f()
{
   Lock(this);
   if( i == 2 )
     i = 3;
}

void ThreadSafe::g()
{
   Lock(this);
   if( i == 3 )
     i = 2;
}
```

Protected-Konstruktoren

`TSync();`
Standardkonstruktor

`TSync( const TSync& );`
Kopierkonstruktor. Das *TCriticalSection*-Objekt wird nicht kopiert.

Protected-Operatoren

operator =

`const TSync& operator = ( const TSync& s );`
s wird dem *TSync*-Objekt zugewiesen. Das *TCriticalSection*-Objekt wird nicht kopiert.

Klasse TSync::Lock — thread.h

```
class Lock: private TCriticalSection::Lock
```

Diese geschachtelte Klasse verwaltet das Sperren und Entsperren von kritischen Bereichen.

Konstruktoren

```
Lock( const TSync *s);
```

Fordert die Sperrung des kritischen Bereichs des *TSync*-Objekts an, auf das *s* zeigt. Falls das *TCriticalSection*-Objekt von keinen anderen *Lock*-Objekt gesperrt ist, wird der Anforderung stattgegeben und die Ausführung fortgesetzt. Andernfalls wird der anfragende Thread angehalten, bis die Sperre wieder aufgehoben wird.

Destruktor

```
~Lock();
```

Hebt die Sperrung wieder auf.

Klasse TThread — thread.h

```
class TThread
```

TThread realisiert eine systemunabhängige Schnittstelle zu Threads. Hierzu ein Beispiel:

```
class TimerThread : private TThread
{
public:
   TimerThread() : Count(0) {}
private:
   unsigned long Run();
   int Count;
};

unsigned long TimerThread::Run()
{
   // 10 Schleifendurchlaeufe
   while( Count++  10 )
  {
      Sleep(1000);    // 1 Sekunde Pause
      cout < "Durchlauf " < Count < endl;
   }
```

```
    return 0L;
}

int main()
{
    TimerThread timer;
    timer.Start();
    Sleep( 20000 );     // 20 Sekunden Pause
    return 0;
}
```

Typdefinitionen

`enum Status { Created, Running, Suspended, Finished, Invalid };`

Beschreibt folgendermaßen die Zustände des Threads:

- *Created* - das Objekt wurde erzeugt, sein Thread jedoch noch nicht gestartet. Der einzige gültige Folgezustand ist *Running*. Dies wird durch einen Aufruf der Elementfunktion *Start* erreicht. Ein Aufruf von *Suspend* oder *Resume* im Zustand *Created* ist daher ein Fehler, der eine Exception auslöst.
- *Running* - der Thread wurde erfolgreich gestartet. Es gibt zwei mögliche Folgezustände:
- Wenn *Suspend* aufgerufen wird, wechselt das Objekt in den Zustand *Suspended*.
- Wenn der Thread beendet wird, geht das Objekt in den Zustand *Finished* über.

Der Aufruf von *Resume* für ein Objekt im Zustand *Running* ist ein Fehler, der eine Exception auslöst.

- *Suspended* - der Thread wurde angehalten. Weitere Aufrufe von *Suspend* werden gespeichert, so daß entsprechend viele *Resume*-Aufrufe erfolgen müssen, um die Ausführung des Thread fortzusetzen.
- *Finished* - die Thread-Ausführung wurde beendet. Zu diesem Zustand gibt es keinen gültigen Folgezustand. Es ist der einzige Zustand, in dem der Objekt-Destruktor aufgerufen werden darf. Ein Destruktor-Aufruf in irgendeinem anderen Zustand ist ein Fehler, der eine Exception auslöst.

Protected-Konstruktoren

`TThread();`

Erzeugt ein Objekt von Typ *TThread*.

```
TThread( const TThread& );
```
Kopierkonstruktor. Das Objekt befindet sich danach im Zustand *Created*.

Protected-Destruktor

```
virtual ~TThread();
```
Entfernt das *TThread*-Objekt.

Public-Elementfunktionen

GetPriority

```
int GetPriority() const;
```
Ermittelt die Priorität des Thread. Siehe auch *SetPriority*.

GetStatus

```
Status GetStatus() const;
```
Liefert den aktuellen Zustand des Thread zurück. In der Beschreibung des Datenelements *Status* finden Sie die möglichen Rückgabewerte.

Resume

```
unsigned long Resume();
```
Setzt die Ausführung eines angehaltenen Thread fort.

SetPriority

```
int SetPriority( int );
```
Legt die Priorität des Thread fest. Siehe auch *GetPriority*.

Start

```
HANDLE Start();
```
Startet die Ausführung des Thread und liefert das Thread-Handle zurück.

Suspend

```
unsigned long Suspend();
```
Hält die Ausführung des Thread an.

Terminate

```
void Terminate();
```
Setzt ein internes Flag, daß die Ausführung des Thread beendet werden soll. Die abgeleitete Klasse kann den Zustand dieses Flag durch einen Aufruf von *ShouldTerminate* überprüfen.

TerminateAndWait

```
void TerminateAndWait( unsigned long timeout = (unsigned long)(-1) );
```
Ein Kombination von *Terminate* und *WaitForExit*. Das interne Flag für die Beendigung der Thread-Ausführung wird gesetzt, und das *Thread*-Objekt wird blockiert, bis der interne Thread beendet wird oder die mit dem Parameter *timeout* angegebene Zeit (in Millisekunden) abgelaufen ist. Bei einem *timeout*-Wert von -1 wird keine Zeitschranke berücksichtigt.

WaitForExit

```
void WaitForExit( unsigned long timeout = (unsigned long)(-1) );
```
Blockiert das *Thread*-Objekt bis der interne Thread beendet wird oder die mit dem Parameter *timeout* angegebene Zeit (in Millisekunden) abgelau-

fen ist. Bei einem *timeout*-Wert von -1 wird keine Zeitschranke berücksichtigt.

Protected-Elementfunktionen

ShouldTerminate

```
int ShouldTerminate() const;
```

Ein Rückgabewert ungleich null bedeutet, daß zuvor *Terminate* oder *TerminateAndWait* aufgerufen wurden und die Thread-Ausführung beendet werden wird.

Protected-Operatoren

operator =

```
const TThread& operator = ( const TThread& );
```

Die Zuweisungoperator für *TThread*-Objekte. Das *TThread*-Objekt muß sich entweder im Zustand *Created* oder *Finished* befinden, da ansonsten eine Exception ausgelöst wird. Nach erfolgreicher Zuweisung befindet sich das *TThread*-Objekt im Zustand *Created*.

Klasse TThread::TThreadError — thread.h

```
class TThreadError
```

Die Klasse *TThreadError* definiert die Exceptions, die bei Thread-Fehlern ausgelöst werden.

Typdefinitionen

ErrorType

```
enum ErrorType
   {
   SuspendBeforeRun,
   ResumeBeforeRun,
   ResumeDuringRun,
   SuspendAfterExit,
   ResumeAfterExit,
   CreationFailure,
   DestroyBeforeExit,
   AssignError
   };
```

Kennzeichnet den Typ des aufgetretenen Fehlers. In der folgenden Liste werden alle Fehlertypen beschrieben:

- *SuspendBeforeRun* - Aufruf von *Suspend* vor dem Aufruf von *Start*.
- *ResumeBeforeRun* - Aufruf von *Resume* vor dem Aufruf von *Start*.

- *ResumeDuringRun* - Aufruf von *Resume* für einen Thread, der nicht im angehaltenen Zustand *(Suspended*) war.
- *SuspendAfterExit* - Aufruf von *Suspend* für Objekt, dessen Thread bereits beendet worden war.
- *ResumeAfterExit* - Aufruf von *Resume* für Objekt, dessen Thread bereits beendet worden war.
- *CreationFailure* - Das Betriebssystem konnte den Thread nicht erzeugen.
- *DestroyBeforeExit* - Der Objekt-Destruktor wurde vor der Beendigung der Thread-Ausführung aufgerufen.
- *AssignError* - Es wurde versucht, ein Objekt zuzuweisen, das sich nicht im Zustand *Created* oder *Finished* befand.

Public-Elementfunktionen

GetErrorType

```
ErrorType GetErrorType() const;
```

Liefert den Typ des aufgetretenen Fehlers zurück.

Typdefinitionen für TTime time.hTTime;Typdefinitionen für

```
typedef unsigned HourTy;
typedef unsigned MinuteTy;
typedef unsigned SecondTy;
typedef unsigned long ClockTy;
```

Die Typdefinitionen für Stunden, Minuten, Sekunden und Sekunden seit dem 1. Januar 1901.

Klasse TTime time.h

```
class TTime
```

Die Klasse *TTime* kapselt Zeitangaben und Zeitfunktionen.

Protected-Datenelemente

RefDate

```
static const TDate RefDate;
```

Die früheste gültige Datum für *TTime*-Objekte: 1. Januar 1901.

MaxDate

```
static const TDate MaxDate;
```

Das späteste gültige Datum für *TTime*-Objekte.

Konstruktoren

`TTime();`
Erzeugt ein *TTime*-Objekt mit dem aktuellen Datum.

`TTime( ClockTy s );`
Erzeugt ein *TTime*-Objekt aus dem angegebenen Parameter *s* (Sekunden seit dem 1. Januar 1901).

`TTime( HourTy h, MinuteTy m, SecondTy s = 0 );`
Erzeugt ein *TTime*-Objekt mit der angegebenen Zeit und dem Tagesdatum.

`TTime( const TDate&, HourTy h=0, MinuteTy m=0, SecondTy s=0 );`
Erzeugt ein *TTime*-Objekt mit der angegebenen Zeit und dem angegebenen Datum.

Public-Elementfunktionen

AsSting
`string AsString() const;`
Liefert einen *string*-Objekt zurück, das die Zeitangabe enthält.

BeginDST
`static TTime BeginDST( unsigned year );`
Liefert den Beginn der Sommerzeit für das angegebene Jahr zurück.

Between
`int Between( const TTime& a, const TTime& b ) const;`
Liefert den Wert 1 zurück, falls das Datum des *TTime*-Objekts zwischen *a* und *b* liegt. Andernfalls wird 0 zurückgeliefert.

CompareTo
`int CompareTo( const TTime & t ) const;`
Vergleicht *t* mit dem *TTime*-Objekt und liefert den Wert 0 zurück, falls die beiden Zeitangaben gleich sind. Liegt *t* zeitlich vorher, so wird 1, liegt es nachher, so wird -1 zurückgeliefert.

EndDST
`static TTime EndDST( unsigned year );`
Liefert das Ende der Sommerzeit für das angegebene Jahr zurück.

Hash
`unsigned Hash() const;`
Liefert die Sekunden seit dem 1. Januar 1901 zurück.

Hour
`HourTy Hour() const;`
Liefert den Stundenwert in lokaler Zeit zurück.

HourGMT
`HourTy HourGMT() const;`
Liefert den Stundenwert in Greenwich Mean Time (GMT) zurück.

IsDST

```
int IsDST() const;
```

Liefert den Wert 1 zurück, wenn die Sommerzeit gültig ist. Andernfalls wird 0 zurückgeliefert.

IsValid

```
int IsValid() const;
```

Liefert den Wert 1 zurück, wenn das *TTime*-Objekt eine gültige Zeit enthält. Andernfalls wird 0 zurückgeliefert.

Max

```
TTime Max( const TTime& t ) const;
```

Liefert entweder das *TTime*-Objekt oder *t* zurück, je nachdem, welches größer ist.

Min

```
TTime Min( const TTime& t ) const;
```

Liefert entweder das *TTime*-Objekt oder *t* zurück, je nachdem, welches kleiner ist.

Minute

```
MinuteTy Minute() const;
```

Liefert den Minutenwert in lokaler Zeit zurück.

MinuteGMT

```
MinuteTy MinuteGMT() const;
```

Liefert den Minutenwert in Greenwich Mean Time (GMT) zurück.

PrintDate

```
static int PrintDate( int flag);
```

Enthält *flag* den Wert 1, so wird das Datum zusammen mit der Zeit ausgegeben. Enthält *flag* den Wert 0, so wird das Datum nicht ausgegeben. Die alte Einstellung wird zurückgeliefert.

Second

```
SecondTy Second() const;
```

Liefert den Sekundenwert zurück.

Seconds

```
ClockTy Seconds() const;
```

Liefert die Sekunden seit dem 1. Januar 1901 zurück.

Protected-Elementfunktionen

AssertDate

```
static int AssertDate( const TDate& d );
```

Liefert den Wert 1 zurück, falls *d* zwischen dem frühesten (*Refdate*) und dem spätesten gültigen Datum (*Maxdate*) liegt.

Operatoren

operator <

```
int operator < ( const TTime& t ) const;
```

Liefert den Wert 1 zurück, wenn das *TTime*-Objekt kleiner als der Parameter *t* ist. Andernfalls wird 0 zurückgeliefert.

operator <=

```
int operator <= ( const TTime& t ) const;
```

Liefert den Wert 1 zurück, wenn das *TTime*-Objekt kleiner oder gleich *t* ist. Andernfalls wird 0 zurückgeliefert.

operator >

```
int operator > ( const TTime& t ) const;
```

Liefert den Wert 1 zurück, wenn das *TTime*-Objekt größer als der Parameter *t* ist. Andernfalls wird 0 zurückgeliefert.

operator >=

```
int operator >= ( const TTime& t ) const;
```

Liefert den Wert 1 zurück, wenn das *TTime*-Objekt größer oder gleich *t* ist. Andernfalls wird 0 zurückgeliefert.

operator ==

```
int operator == ( const TTime& t ) const;
```

Liefert den Wert 1 zurück, wenn das *TTime*-Objekt gleich *t* ist. Andernfalls wird 0 zurückgeliefert.

operator !=

```
int operator != ( const TTime& t ) const;
```

Liefert den Wert 1 zurück, wenn das *TTime*-Objekt ungleich *t* ist. Andernfalls wird 0 zurückgeliefert.

operator ++

```
void operator++();
```

Inkrementiert die Zeit um eine Sekunde.

operator --

```
void operator--();
```

Dekrementiert die Zeit um eine Sekunde.

operator +=

```
void operator+=(long s);
```

Erhöht den Zeitwert um *s* Sekunden.

operator -=

```
void operator-=(long s);
```

Erniedrigt den Zeitwert um *s* Sekunden.

operator +

```
friend TTime operator + ( const TTime& t, long s );
friend TTime operator + ( long s, const TTime& t );
```

Erhöht den Zeitwert um *s* Sekunden.

operator -

```
friend TTime operator - ( const TTime& t, long s );
friend TTime operator - ( long s, const TTime& t );
```

Führt eine Subtraktion mit den Parametern *s* und *t* durch.

operator <<

```
friend ostream& operator << ( ostream& os, const TTime& t);
```

Schreibt *t* in den Ausgabe-Stream *os*.

```
friend opstream& operator << ( opstream& s, const TTime& d );
```

Schreibt *d* in den persistenten Stream *s*.

operator >>

```
friend ipstream& operator >> ( ipstream& s, TTime& d );
```

Liest *d* aus dem persistenten Stream *s*.

Index

0x11, BIOS-Interrupt *57*, *58*, *59*
0x12, BIOS-Interrupt *60*
0x16, BIOS-Interrupt *59*
0x19, DOS-Funktion *92*, *145*
0x1A, BIOS-Interrupt *61*
0x21, DOS-Interrupt *165*
0x23, DOS-Interrupt *82*
0x33, DOS-Systemaufruf *140*, *250*
0x3F, DOS-Funktion *239*
0x40, DOS-Funktion *99*
0x44, DOS-Systemaufruf *167*
0x4E, DOS-Funktion *89*
0x4E, DOS-Systemaufruf *118*
0x59, DOS-Systemaufruf *88*
0x62, DOS-Systemaufruf *149*
32-Bit Programme *41*
8086-Prozessor
 Interrupt *163*, *164*
 Interrupt-Vektoren *95*, *98*, *152*
 Interrupts *166*
8087-Koprozessor
 siehe mathematischer Koprozessor *325*
80x86
 Funktionen des *32*
80x87
 siehe Mathematischer Coprozessor *126*
_8087 (globale Variable) *325*
__DELTA (Makro) *377*
__emit__ (Bibliotheksfunktion) *102*
__throwExeceptionName (globale Variable) *333*
__throwFileName (globale Variable) *333*
__throwLineNumber (globale Variable) *333*
_argc (globale Variable) *326*
_argv (globale Variable) *326*
_atold (Bibliotheksfunktion) *51*
_beginthread (Bibliotheksfunktion) *55*, *56*
_bios_equiplist (Bibliotheksfunktion) *58*
_bios_memsize (Bibliotheksfunktion) *60*
_bios_timeofday (Bibliotheksfunktion) *61*
_c_exit (Bibliotheksfunktion) *65*
_cexit (Bibliotheksfunktion) *65*
_chain_intr(Bibliotheksfunktion) *66*
_chdrive (Bibliotheksfunktion) *68*
_crotl (Bibliotheksfunktion) *80*
_crotr (Bibliotheksfunktion) *80*
_ctype (globale Variable) *326*
_daylight (globale Variable) *327*
_directvideo (globale Variable) *327*
_dos_close (Bibliotheksfunktion) *86*
_dos_commit (Bibliotheksfunktion) *86*
_dos_creat (Bibliotheksfunktion) *86*
_dos_creatnew (Bibliotheksfunktion) *87*
_dos_findfirst (Bibliotheksfunktion) *89*
_dos_findnext (Bibliotheksfunktion) *90*
_dos_getdate (Bibliotheksfunktion) *91*
_dos_getdiskfree (Bibliotheksfunktion) *92*
_dos_getdrive (Bibliotheksfunktion) *92*
_dos_getfileattr (Bibliotheksfunktion) *93*
_dos_getftime (Bibliotheksfunktion) *93*
_dos_gettime (Bibliotheksfunktion) *94*
_dos_getvect (Bibliotheksfunktion) *95*
_dos_open (Bibliotheksfunktion) *95*
_dos_read (Bibliotheksfunktion) *97*
_dos_setdate (Bibliotheksfunktion) *91*
_dos_setdrive (Bibliotheksfunktion) *92*
_dos_setfileattr (Bibliotheksfunktion) *93*
_dos_setftime (Bibliotheksfunktion) *93*
_dos_settime (Bibliotheksfunktion) *94*
_dos_setvect (Bibliotheksfunktion) *98*
_dos_write (Bibliotheksfunktion) *99*
_doserrno (globale Variable) *328*
_endthread(Bibliotheksfunktion) *103*
_environ (globale Variable) *327*
_exit (Bibliotheksfunktion) *107*
_expand (Bibliotheksfunktion) *109*
_floatconvert (globale Variable) *330*
_fmemccpy (Bibliotheksfunktion) *194*
_fmemchr (Bibliotheksfunktion) *194*
_fmemcmp (Bibliotheksfunktion) *195*
_fmemcpy (Bibliotheksfunktion) *195*
_fmemicmp (Bibliotheksfunktion) *196*
_fmemset (Bibliotheksfunktion) *197*
_fpreset (Bibliotheksfunktion) *126*
_fsopen (Bibliotheksfunktion) *133*
_fstrcat (Bibliotheksfunktion) *275*
_fstrchr (Bibliotheksfunktion) *275*

_fstrcmp (Bibliotheksfunktion) *276*
_fstrcpy (Bibliotheksfunktion) *277*
_fstrcspn (Bibliotheksfunktion) *278*
_fstrdup (Bibliotheksfunktion) *278*
_fstricmp (Bibliotheksfunktion) *282*
_fstrlen (Bibliotheksfunktion) *283*
_fstrlwr (Bibliotheksfunktion) *283*
_fstrncat (Bibliotheksfunktion) *284*
_fstrncmp (Bibliotheksfunktion) *284*
_fstrncpy (Bibliotheksfunktion) *285*
_fstrnicmp (Bibliotheksfunktion) *286*
_fstrnset (Bibliotheksfunktion) *286*
_fstrpbrk (Bibliotheksfunktion) *287*
_fstrrchr (Bibliotheksfunktion) *287*
_fstrrev (Bibliotheksfunktion) *288*
_fstrset (Bibliotheksfunktion) *288*
_fstrspn (Bibliotheksfunktion) *289*
_fstrstr (Bibliotheksfunktion) *289*
_fstrtok (Bibliotheksfunktion) *291*
_fstrupr (Bibliotheksfunktion) *294*
_fullpath (Bibliotheksfunktion) *138*
_get_osfhandle (Bibliotheksfunktion) *149*
_getdcwd (Bibliotheksfunktion) *143*
_heapadd (Bibliotheksfunktion) *156*
_heapchk (Bibliotheksfunktion) *157*
_HEAPEMPTY *237*
_HEAPEND *159, 237*
_heapmin (Bibliotheksfunktion) *158*
_HEAPOK *159*
_heapset (Bibliotheksfunktion) *159*
_InitEasyWin (Bibliotheksfunktion) *161*
_lrotl (Bibliotheksfunktion) *185*
_makepath (Bibliotheksfunktion) *187*
_matherrl (Bibliotheksfunktion) *189*
_msize (Bibliotheksfunktion) *202*
_new_handler (globale Variable) *331*
_open (Bibliotheksfunktion) *203*
_open_osfhandle (Bibliotheksfunktion) *206*
_osmajor (globale Variable) *332*
_osminor (globale Variable) *332*
_pipe (Bibliotheksfunktion) *211*
_popen (Bibliotheksfunktion) *213*
_psp (globale Variable) *332*
_rotl (Bibliotheksfunktion) *234*
_rotr (Bibliotheksfunktion) *234*
_rtl_chmod (Bibliotheksfunktion) *234*
_rtl_close (Bibliotheksfunktion) *235*
_rtl_creat (Bibliotheksfunktion) *236*
_rtl_heapwalk (Bibliotheksfunktion) *236*
_rtl_open (Bibliotheksfunktion) *237*
_rtl_read (Bibliotheksfunktion) *239*
_rtl_write (Bibliotheksfunktion) *239*
_searchenv (Bibliotheksfunktion) *247*
_searchstr (Bibliotheksroutine) *248*
_set_new_handler (Bibliotheksfunktion) *331*
_setcursortype (Bibliotheksfunktion) *251*
_splitpath (Bibliotheksfunktion) *270*
_status87 (Bibliotheksfunktion) *273*
_strdate (Bibliotheksfunktion) *278*
_strerror (Bibliotheksfunktion) *279*
_strtime (Bibliotheksfunktion) *289*
_strtold (Bibliotheksfunktion) *290*
_sys_errlist (globale Variable) *328*
_sys_nerr (globale Variable) *328*
_threadid (globale Variable) *333*
_timezone (globale Variable) *307*
 Setzen *307*
_tolower (Bibliotheksmakro) *305*
_toupper (Bibliotheksmakro) *306*
_tzname (globale Variable *307*
_tzname (globale Variable)
 Setzen *307*
_version (globale Variable) *334*
_wscroll (globale Variable) *335*

A

Abbruch
 von Programmen *45, 107*
 von Programmen mit assert *49*
abort (Bibliotheksfunktion) *45*
Abrunden *121*
abs
 complex-Elementfunktion *493*
abs (Bibliotheksmakro) *45*
absolute Werte
 von Gleitkommazahlen *109*
Absolutwerte *109, 175*
 Integer *45*
 komplexe Zahlen und *63*
access (Bibliotheksfunktion) *46*
acos
 complex-Elementfunktion *493*
acos (Bibliotheksfunktion) *46*
Add
 TBinarySearchTreeImp-Elementfunktion *408*
 TIBinarySearchTreeImp-Elementfunktion *410*

TMArrayAsVector-Elementfunktion *384*
TMBagAsVector-Elementfunktion *403*
TMCVectorImp-Elementfunktion *475*
TMDictionaryAsHashTable-Elementfunktion *423*
TMDoubleListImp-Elementfunktion *430*
TMHashTableImp-Elementfunktion *439*
TMIArrayAsVector-Elementfunktion *390*
TMIBagAsVector-Elementfunktion *405*
TMICVectorImp-Elementfunktion *483*
TMIDictionaryAsHashTable-Elementfunktion *426*
TMIDoubleListImp-Elementfunktion *434*
TMIHashTableImp-Elementfunktion *441*
TMIListImp-Elementfunktion *449*
TMISetAsVector-Elementfunktion *462*
TMListImp-Elementfunktion *445*
TMSetAsVector-Elementfunktion *460*

AddAt
- TMArrayAsVector-Elementfunktion *384*
- TMCVectorImp-Elementfunktion *475*
- TMIArrayAsVector-Elementfunktion *390*

AddAtHead
- TMIDoubleListImp-Elementfunktion *434*

AddAtTail
- TMDoubleListImp-Elementfunktion *430*
- TMIDoubleListImp-Elementfunktion *435*

Adreßsegment
- far-Zeiger *126*

Adressen
- an __emit__ übergebene *102*
- far *126*
- Speicher *102*

alloc.h (Header-Datei) *26*
alloca (Bibliotheksfunktion) *47*

allocate
- streambuf-Elementfunktion *357*

Alphabetische Zeichen
- prüfen auf *169*

Alphanumerische Zeichen
- prüfen auf *169*

ansi_to_oem
- String-Elementfunktion *520*

append
- String-Elementfunktion *520*

Arcuscosinus *46*
Arcussinus *48*
Arcustangens *49, 50*

argc
- Parameter für main *37*

argkomplexe Zahlen
- Winkel *493*

ARGS.EXE *38*

Argumentlisten
- variabler Länge *36*

argv
- Parameter für main *37*

arithmetische Routinen
- _lrotl *185*
- _rotl *234*
- _rotr *234*
- div *85*
- ldiv *176*

Array
- Elemente sortieren *226*

Arrays
- absuchen *62*
- durchsuchen *176*
- Sortieren *226*
- Zeichenattribute *326*

ArraySize
- TMArrayAsVector-Elementfunktion *384*
- TMIArrayAsVector-Elementfunktion *390*

ASCII-Format
- Zeichen kovertieren in *305*

ASCII-Zeichen
- prüfen *169, 170, 171, 172, 173, 174*

asctime (Bibliotheksfunktion) *48*

asin
- complex-Elementfunktion *493*

asin (Bibliotheksfunktion) *48*
asinl (Bibliotheksfunktion) *48*
assert (Bibliotheksmakro) *49*
assert.h (Header-Datei) *26*

assert_element
- String-Elementfunktion *527*
- TSubString -Elementfunktion *532*

assert_index
- String-Elementfunktion *527*

AssertDate
- TTime-Elementfunktion *543*

AssertIndexOfMonth
- TDate-Elementfunktion *514*

AssertWeekDayNumber
- TDate-Elementfunktion *514*

assign
- String-Elementfunktion *521*

AsSting

TTime-Elementfunktion *542*
AsString
TDate-Elementfunktion *512*
atan
complex-Elementfunktion *493*
atan (Bibliotheksfunktion) *49*
atan2 (Bibliotheksfunktion) *50*
atan2l (Bibliotheksfunktion) *50*
atanl (Bibliotheksfunktion) *49*
atexit (Bibliotheksfunktion) *51*
atof (Bibliotheksfunktion) *51*
atoi (Bibliotheksfunktion) *52*
atol (Bibliotheksfunktion) *53*
attach
filebuf-Elementfunktion *340*
fpbase-Elementfunktion *364*
fstreambase-Elementfunktion *343*
Attribute *235*
Text- *299, 300, 301*
Zeichen *326*
Aufruf
DOS-Kommando *296*
Ausgabe
Fehlermeldungen *328*
formatiert zu stdout *216*
formatierte *315*
Pufferung *259*
siehe auch Ein-/Ausgabe *80, 216, 316*
von Strings *77, 128, 224*
Ausgaben
formatiert in Streams *127*
formatierte *76, 216, 271, 314, 317*

B

bad
ios-Elementfunktion *347*
pstream-Elementfunktion *373*
Bad_case (Exception) *503*
Bad_cast (Klasse) *503*
Bad_typeid (Klasse) *503*
base
streambuf-Elementfunktion *358*
BCD-Zahlen *489*
Konvertierung in long double *492*
bcd.h (Header-Datei) *26*
bdos (Bibliotheksfunktion) *53*
bdosptr (Bibliotheksfunktion) *54*
Bedingung
prüfen *49*
before
Type_info-Elementfunktion *507*
BeginDST
TTime-Elementfunktion *542*
Behandlungsroutinen *82*
Berechunungsfehler
Behandlung *189*
Betriebssystem
Attributwort *87*
Between
TDate-Elementfunktion *512*
TTime-Elementfunktion *542*
Bibliotheken
Auswahl von *22*
DLL *25*
Liste der *23*
Bibliotheksfunktionen *43*
Bibliotheksmakro
inp *161*
inportb *162*
inpw *162*
Bildausschnitte
im Textmodus *150, 224*
kopieren *201*
Bildlauf
_wscroll (glob. Variable) *335*
Bildschirm
Ausgabe auf *142, 327*
direktes Schreiben *327*
formatierte Ausgabe auf *76*
löschen *72*
Strings anzeigen *77*
Text speicher *150*
Zeichen schreiben in *222*
Bildschirmausgabe
gettext *150*
gettextinfo *151*
gotoxy *155*
highvideo *160*
insline *163*
lowvideo *184*
movetext *201*
normvideo *202*
puttext *224*
textattr *299*
textbackground *300*
textcolor *301*

textmode *302*
wherex *320*
wherey *321*
window *321*
Bildschirmmodus *58*
binär
Suche *62*
binär codierte Dezimalzahlen
BCD *489*
Binäre Dateien *304*
_fsopen und *133*
creat und *77*
creattemp und *79*
fdopen *114*
temporär öffnen *304*
binärer Modus *99, 204, 240, 265, 322*
BIOS
Timer *61*
BIOS-Funktionen *32*
BIOS-Interrupt
0x16 *59*
BIOS-Interrupts
0x11 *57, 58, 59*
0x12 *60*
0x1A *61*
bios.h (Header-Datei) *26*
biosequip (Bibliotheksfunktion) *57*
bioskey (Bibliotheksfunktion) *59*
biosmemory (Bibliotheksfunktion) *60*
biostime (Bibliotheksfunktion) *61*
bitalloc
ios-Elementfunktion *347*
Bitrotation *80*
long Integer *185*
bitweise Rotation *185, 234*
blen
streambuf-Elementfunktion *358*
block-orientierte Dateien *250, 258*
Blockoperationen
siehe auch Editieren *299*
BoundBase
TMArrayAsVector-Elementfunktion *386*
TMIArrayAsVector-Elementfunktion *392*
bp
pstream-Elementfunktion *374*
bsearch (Bibliotheksfunktion) *62*
Byte
aus Schnittstelle lesen *162*
Bytes
lesen aus Speicher *209*
Positionen vertauschen *295*
speichern *212*
Zeichen aus mehreren *192, 193*

C

C++
Speicherreservierung *331*
c_str
String-Elementfunktion *521*
cabs (Bibliotheksfunktion) *63*
cabsl (Bibliotheksfunktion) *63*
calloc (Bibliotheksfunktion) *64*
Carry-Flag *164, 165, 166*
CastableID
TStreamableBase-Elementfunktion *375*
ceil (Bibliotheksfunktion) *64*
ceill (Bibliotheksfunktion) *64*
cgets (Bibliotheksfunktion) *66*
chdir (Bibliotheksfunktion) *67*
CHECK (Diagnose-Makro) *498*
checks.h (Header-Datei) *26*
CHECKX (Diagnose-Makro) *500*
child process *105*
child-Prozess *267*
chmod (Bibliotheksfunktion) *68*
clear
ios-Elementfunktion *347*
pstream-Elementfunktion *373*
clear87_ (Bibliotheksfunktion) *69*
clearerr (Bibliotheksfunktion) *70*
CLK_TCK
Konstante *70*
CLK_TCK (Konstante) *70*
clock (Bibliotheksfunktion) *70*
clock_t
Datentyp *70*
close
filebuf-Elementfunktion *341*
fpbase-Elementfunktion *364*
fstreambase-Elementfunktion *344*
TFile-Elementfunktion *517*
close (Bibliotheksfunktion) *71*
closedir (Bibliotheksfunktion) *71*
clreol
conbuf-Elementfunktion *338*
clreol (Bibliotheksfunktion) *72*

clrscr
 conbuf-Elementfunktion *338*
 constream-Elementfunktion *339*
clrscr (Bibliotheksfunktion) *72*
COMMAND.COM *296*
compare
 String-Elementfunktion *521*
CompareTo
 TDate-Elementfunktion *512*
 TTime-Elementfunktion *542*
complex (Klasse) *492*
complex.h (Header-Datei) *26*
COMSPEC (Environment-Variable *296*
conbuf (Klasse) *337*
CondFunc
 TMArrayAsVector-Funktionstyp *384*
 TMBagAsVector-Funktionstyp *402*
 TMDequeAsDoubleList-Funktionstyp *418*
 TMDequeAsVector-Funktionstyp *412*
 TMDoubleListImp-Funktionstyp *429*
 TMIArrayAsVector-Funktionstyp *389*
 TMIBagAsVector-Funktionstyp *405*
 TMIDequeAsDoubleList-Funktionstyp *420*
 TMIDequeAsVector-Funktionstyp *415*
 TMIDoubleListImp-Funktionstyp *434*
 TMIListImp-Funktionstyp *449*
 TMIStackAsVector-Funktionstyp *466*
 TMIVectorImp-Funktionstyp *479*
 TMListImp-Funktionstyp *445*
 TMStackAsVector-Funktionstyp *463*
 TMVectorImp-Funktionstyp *471*
conio.h (Header-Datei) *26*
conj
 complex-Elementfunktion *493*
constrea.h (Header-Datei) *26*
constream (Klasse) *339*
contains
 String-Elementfunktion *521*
control87_ (Bibliotheksfunktion) *73*
copy
 String-Elementfunktion *521*
cos
 complex-Elementfunktion *493*
cos (Bibliotheksfunktion) *74*
cosh
 complex-Elementfunktion *493*
cosh (Bibliotheksfunktion) *74*
coshl (Bibliotheksfunktion) *74*
Cosinus *46, 74*
cosl (Bibliotheksfunktion) *74*
Count
 TMCVectorImp-Elementfunktion *475*
Count_
 TMCVectorImp-Datenelement *476*
country (Bibliotheksfunktion) *75*
cow
 String-Elementfunktion *527*
cprintf (Bibliotheksfunktion) *76*
cputs (Bibliotheksfunktion) *77*
CR/LF-Zeichenfolge *240, 322*
CR/LF-Zeichenpaar *99*
creat (Bibliotheksfunktion) *77*
creatnew (Bibliotheksfunktion) *78*
creattemp (Bibliotheksfunktion) *79*
cscanf (Bibliotheksfunktion) *80*
cstring.h (Header-Datei) *26*
ctime (Bibliotheksfunktion) *81*
ctrlbrk (Bibliotheksfunktion) *82*
ctype.h (Header-Datei) *26*
Current
 TBinarySearchTreeIteratorImp-
 Elementfunktion *409*
 TIBinarySearchTreeIteratorImp-
 Elementfunktion *411*
 TMArrayAsVectorIterator-
 Elementfunktion *388*
 TMDequeAsVectorIterator-
 Elementfunktion *414*
 TMDictionaryAsHashTableIterator-
 Elementfunktion *424*
 TMDoubleListIteratorImp-
 Elementfunktion *431*
 TMHashTableIteratorImp-
 Elementfunktion *440*
 TMIArrayAsVectorIterator-
 Elementfunktion *393*
 TMIDictionaryAsHashTableIterator-
 Elementfunktion *427*
 TMIDoubleListIteratorImp-
 Elementfunktion *436*
 TMIHashTableIteratorImp-
 Elementfunktion *442*
 TMIListIteratorImp-Elementfunktion *450*
 TMIVectorIteratorImp-Elementfunktion *481*
 TMListIteratorImp-Elementfunktion *447*
 TMVectorIteratorImp-Elementfunktion *473*
Cursor
 Form festlegen *251*

Position ermitteln *320, 321*
Position im Textfenster *155*
Position setzen *155*
cwait (Bibliotheksfunktion) *82*

D

Data
TMDequeAsVector-Datenelement *413*
TMDoubleListElement-Datenelement *428*
TMListElement-Datenelement *444*
Datei
Größe ändern *69*
Modus *87*
Zugriffsmodus *309*
Datei-Handle *86*
Stream zuordnen *113*
verdoppeln *100, 101*
Dateiattribute
DOS *93*
setzen *87*
suchen *118*
Dateibearbeitung
Attribute *77, 79, 86, 236*
Datum/Uhrzeit lesen *147*
Erzeugen von Dateien *77, 78, 79, 86, 87, 236*
gemeinsamer Zugriff *180*
Lesen und Schreiben *129*
mktemp *199*
Öffnen *125*
Positionierung *186, 232, 297*
SHARED-Modus *181*
Suche *89, 90*
Suchwege analysieren/konstruieren *123, 124, 187, 270*
Zugriffsberechtigung ändern *68, 234*
Zugriffsberechtigung prüfen *46*
Dateien
absuchen *120*
ARGS.EXE *38*
Attribute *68, 77, 79, 234*
binäre *114, 257*
binärer Modus *204, 265*
Dateiende-Bestimmung *104, 115*
Datum und Uhrzeit *93*
Datum/Uhrzeit setzen *312*
Datum/Uhrzeit und setzen *147*
eindeutige Namen *298*
erzeugen *77, 86, 87, 95, 203, 236, 237*
formatierte Ausgabe *127, 216, 271, 314, 315, 317*
formatierte Eingabe *80, 240, 272, 314, 316, 317*
Handle *113*
Handle verdoppeln *100, 101*
Informationen über *135*
Längenbestimmung *117*
lesen *77*
lesen aus *97, 229, 239*
lesen aus Streams *117*
lesen und formatieren *131, 240, 314, 316*
Lesen/Schreiben-Zugriff *77*
Löschen *231, 311*
Modus *77, 78*
Namensanalyse *208*
neue *79*
offene *135*
öffnen *95, 125, 203, 205, 237, 265*
öffnen zur Aktualisierung *114*
Position ermitteln *116*
Positionierung *137*
Referenznummer *118*
schließen *71, 86, 112, 130, 235*
schreiben in *99, 139, 239, 322*
SHARED-Modus *96, 133, 238, 265, 311*
Sperre aufheben *311*
sperren *180, 181*
Streams öffnen *125*
suchen *247, 248*
temporäre *298*
temporäre benennen *304*
temporäre löschen *234*
temporäre öffnen *304*
Textmodus *204, 265*
überschreiben *77, 86, 236*
umbenennen *232*
Verzeichnis absuchen *118*
WILDARGS.OBJ *39*
Zeichen lesen aus *116, 140*
Zeiger *116*
Zeiger neu setzen *132*
Zeiger setzen *133*
Zugriff auf *46*
Zugriffsberechtigung *68, 234*
Zugriffsmodus *114, 125, 130, 133, 135*
Zugriffsmodus ändern *93*

Dateiende-Flag *70*
Dateiende-Indikator *104, 229*
Dateimodus
 setzen *257*
Dateinamen
 eindeutige *304*
 eindeutigen erzeugen *199*
Dateizeiger *229, 239*
 aktuelle Position ermitteln *297*
 aktuelle Position setzen *186*
 initialisieren *232*
 setzen *266*
 zurückliefern *137*
Daten
 kopieren *200*
 lesen aus Streams *131, 314, 316*
Datensegment *64*
Datensegmente *188*
Datensicherheit *148*
Datentypen
 Bibliothek *63, 148*
 clock_t *70*
 mexcep_ *190*
 sig_atomic_t *261*
 struct COUNTRY *75*
 struct dfree *144*
 struct diskfree_t *92*
 struct div_t *85*
 struct DOSERROR *88*
 struct dostime_t *95*
 struct exception *190*
 struct fatinfo *146, 147*
 struct ffblk *118*
 struct fint_t *89*
 struct REGPACK *167*
 struct stat *135*
 struct text_info *151*
 struct time *152*
 struct timeb *137*
 struct tm *154, 180*
 va_list *313*
Datum
 konvertieren in String *278*
 System- *81*
Datum um Uhrzeit
 Landesspezifische Formate *75*
Datum und Uhrzeit
 _daylight, _timezone (Variablen) *327*
 _dos_getftime *93*
 _dos_gettime *94*
 _strdate *278*
 _strtime *289*
 asctime *48*
 BIOS-Timer *61*
 ctime *81*
 Datei *147*
 difftime *84*
 getftime *147*
 gettime *152*
 globale Variablen *307*
 GMT *154*
 in Kalenderformat konvertieren *199*
 in UNIX-Format umwandeln *99*
 konvertieren *179*
 lesen und setzen *152*
 localtime *179*
 MEZ *82*
 setzen und lesen *93*
 Sommerzeit *327*
 stime *274*
 System *94, 137, 179, 274, 303*
 System- *99*
 UNIX- zu DOS-Format *310*
 vergangene CPU-Zeit seit Aufruf *70*
 von Dateien *312*
Datum- und Uhrzeit
 System- *48*
Datumsangaben
 Funktionen *36*
Day
 TDate-Elementfunktion *512*
daylight (globale Variable) *307*
 setzen *307*
DayOfMonth
 TDate-Elementfunktion *513*
DayOfWeek
 TDate-Elementfunktion *513*
DaysInYear
 TDate-Elementfunktion *513*
DayTy
 TDate-Datentyp *511*
DayWithinMonth
 TDate-Elementfunktion *513*
de_exterror *88*
Debuggen *497, 499, 501*
DECLARE_ABSTRACT_STREAMABLE
 (Makro) *379*

DECLARE_ABSTRACT_STREAMER
 (Makro) *380*
DECLARE_CASTABLE (Makro) *380*
DECLARE_STREAMABLE (Makro) *378*
DECLARE_STREAMABLE_CTOR (Makro) *380*
DECLARE_STREAMABLE_FROM_BASE
 (Makro) *378*
DECLARE_STREAMABLE_OPS (Makro) *380*
DECLARE_STREAMER (Makro) *379*
DECLARE_STREAMER_FROM_BASE
 (Makro) *379*
DeleteNode
 TBinarySearchTreeImp-Element-
 funktion *408*
 TIBinarySearchTreeImp-Element-
 funktion *410*
DeleteType
 TShouldDelete-Datentyp *486*
delline
 conbuf-Elementfunktion *338*
delline (Bibliotheksfunktion) *84*
DelObj
 TShouldDelete-Elementfunktion *487*
Delta
 TMCVectorImp-Datenelement *476*
Destroy
 TMArrayAsVector-Elementfunktion *384*
 TMIArrayAsVector-Elementfunktion *390*
Detach
 TBinarySearchTreeImp-Elementfunktion *408*
 TIBinarySearchTreeImp-Element-
 funktion *410*
 TMArrayAsVector-Elementfunktion *385*
 TMBagAsVector-Elementfunktion *403*

 TMCVectorImp-Elementfunktion *476*
 TMDictionaryAsHashTable-Element-
 funktion *423*
 TMDoubleListImp-Elementfunktion *430*
 TMHashTableImp-Elementfunktion *439*
 TMIArrayAsVector-Elementfunktion *390*
 TMIBagAsVector-Elementfunktion *405*
 TMIDictionaryAsHashTable-Element-
 funktion *426*
 TMIDoubleListImp-Elementfunktion *435*
 TMIHashTableImp-Elementfunktion *441*
 TMIListImp-Elementfunktion *449*
 TMListImp-Elementfunktion *445*
DetachAtHead
 TMIDoubleListImp-Elementfunktion *435*
DetachAtTail
 TMIDoubleListImp-Elementfunktion *435*
DIAG_CREATE_GROUP (Diagnose-Makro)
500
DIAG_DECLARE_GROUP (Diagnose-
 Makro) *500*
DIAG_DEFINE_GROUP (Diagnose-Makro) *500*
DIAG_ENABLE (Diagnose-Makro) *500*
DIAG_GETLEVEL (Diagnose-Makro) *501*
DIAG_ISENABLED (Diagnose-Makro) *500*
DIAG_SETLEVEL (Diagnose-Makro) *501*
Diagnose
 Präprozessorsymbole für *498*
Diagnose-Makros *497, 499, 501*
Diagnose-Routinen
 _matherrl *189*
 matherr *189*
 perror *209*
Diagnoseroutinen
 assert *49*
difftime (Bibliotheksfunktion) *84*
dir.h (Header-Datei) *26*
direct.h (Header-Datei) *26*
dirent.h (Header-Datei) *26*
Disketten
 freien Platz ermitteln *92, 144*
 Sektorenüberprüfung *259*
Diskettenroutinen
 _chdrive *68*
 _dos_getdiskfree *92*
 _getdcwd *143*
 chdir *67*
 getcurdir *142*
 getcwd *142*
 getdfree *144*
 getdisk *145*
 getdta *145*
 getfat *146*
 getverify *153*
 setdisk *145*
 setdta *251*
 setverify *259*
div (Bibliotheksfunktion) *85*
div_t (Datentyp) *85*
Division *122*
 von Integern *85*
Divisionsrest *85, 122, 176*
Divison

Integer *176*
DLL
unterstützte Speichermodelle *25*
DMA (Direct memory access) *57, 58*
doallocate
strstreambuf-Elementfunktion *360*
DOS
Dateiattribute *79, 86, 87, 93, 236*
Environment-Variablen *106, 327*
erweiterter Fehlercode *88*
Fehlernummer (Konstanten) *330*
Pfad durchsuchen *247*
Programmsegmentpräfix *333*
Verify-Flag *153, 259*
Versionsnummer *332, 334*
Verzeichnis löschen *233*
DOS-Dateiattribute *234*
DOS-Environment-Variablen *268*
DOS-Funktionen
0x19 *92, 145*
0x3F *239*
0x40 *99*
0x4E *89*
DOS-Interrupts
0x21 *165*
0x23 *82*
Funktionen *152*
DOS-Kommandos
aufrufen *296*
DOS-Systemaufrufe *54*
0x33 *140, 250*
0x44 *167*
0x4E *118*
0x59 *88*
0x62 *149*
Speichermodelle und *53, 54*
dos.h (Header-Datei) *26*
DOS/BIOS-Routinen
_bios_equiplist *58*
_bios_memsize *60*
_bios_timeofday *61*
_disable *85*
_dos_getdiskfree *92*
_dos_getvect *95*
_dos_setvect *98*
_enable *85*
bdos *53*
bdosptr *54*
biosequip *57*
bioskey *59*
biosmemory *60*
biostime *61*
country *75*
ctrlbrk *82*
disable *85*
dosexterr *88*
enable *85*
FP_OFF *126*
FP_SEG *126*
geninterrupt *139*
getcbrk *140*
getdfree *144*
getdta *145*
getfat *146*
getfatd *147*
getpsp *149*
getvect *152*
getverify *153*
inp *161*
inportb *162*
inpw *162*
int86 *163*
int86x *164*
intdos *165*
intdosx *165*
intr *166*
ioctl *167*
MK_FP *198*
outport *207*
outportb *207*
outpw *207*
parsfnm *208*
peek *209*
peekb *209*
poke *212*
pokeb *212*
segread *249*
setcbrk *250*
setdta *251*
setvect *152*
setverify *259*
sleep *264*
unlink *311*
dosexterr (Bibliotheksfunktion) *88*
dostounix (Bibliotheksfunktion) *99*
Dreieck
Hypotenuse berechnen *160*
Druckbare Zeichen

prüfen auf *172*
Drucker
installierte *57*
Überprüfung auf Vorhandensein *58*
DTA (Disk Transfer Address) *145*
ermitteln *145*
dup (Bibliotheksfunktion) *100*
dup2 (Bibliotheksfunktion) *101*
dynamische Speicherreservierung *273*
dynamische Speicherverwaltung
calloc *64*
Erweitern Heap *156*
FAR-Heap *109, 110*
malloc *188*
realloc *231*
Speicher freigeben *129*
Speicherreservierung auf FAR-Heap *111*
Überprüfung des Heap *156, 157*

E

E2BIG, EACCES...EZERO (Fehler-Konstanten) *329*
eatwhite
istream-Elementfunktion *352*
eback
streambuf-Elementfunktion *358*
ebuf
streambuf-Elementfunktion *358*
ecvt (Bibliotheksfunktion) *101*
Editieren
Blockoperationen *194, 195*
egptr
streambuf-Elementfunktion *358*
Ein-/Ausgabe
aus Streams *240*
formatierte *271, 272*
Funktionen zur *30*
Puffer *249*
Schnittstellen *161, 162*
Steuerung *167*
Streams *114, 125, 130, 133, 309*
Tastatur *142, 175, 310*
Zeichen auf den Bildschirm *222*
zu Schnittstelle *207*
Ein-/Ausgaben
Bildschirm *76*
Tastatur *141*
Einfügen von Bildschirmzeilen *163*
Eingabe
formatierte *80, 240, 272, 314, 316, 317*
Pufferung *259*
siehe auch Ein-/Ausgabe *80, 316*
Eingabefelder von scanf *243*
Elternprozeß *105*
Elternprozesse *104, 267*
EndDST
TTime-Elementfunktion *542*
env
Parameter für main *37*
Environment
erweitern *223*
Environment ('Umgebung')
Lesen *146*
Übergabe *106, 268*

eof
ios-Elementfunktion *347*
pstream-Elementfunktion *373*
eof (Bibliotheksfunktion) *104*
epptr
streambuf-Elementfunktion *358*
EqualTo
TBinarySearchTreeImp-Elementfunktion *408*
TIBinarySearchTreeImp-Elementfunktion *410*
errno (globale Variable) *328*
errno.h (Header-Datei) *26*
ErrorType
TThread::TThreadError-Elementfunktion *540*
Erweiterter Fehlercode von DOS *88*
except.h (Header-Datei) *26*
Exceptions *503*
Bad_cast *503*
Bad_typeid *503*
Behandlung von *503*
Meldungen *508*
set_unexpected (Funktion) *505*
terminate (Funktion) *506*
unexpected (Funktion) *507*
vordefinierte *503, 508*
xalloc *503, 504*
excpt.h (Header-Datei) *26*
execl (Bibliotheksfunktion) *104*
execle (Bibliotheksfunktion) *104*
execlp (Bibliotheksfunktion) *104*
execlpe (Bibliotheksfunktion) *104*
execv (Bibliotheksfunktion) *104*

execve (Bibliotheksfunktion) *104*
execvp (Bibliotheksfunktion) *104*
execvpe (Bibliotheksfunktion) *104*
exit (Bibliotheksfunktion) *107*
Exit-Code *107*
exp
complex-Elementfunktion *494*
exp (Bibliotheksfunktion) *108*
expl (Bibliotheksfunktion) *108*
Exponent
double *175*
einer Gleitkommazahl *131*
Exponenten
Berechnung *214, 215*
Exponentialfunktion *108*

F

FA_ARCH...FA_SYSTEM (Konstanten) *119*
fabs (Bibliotheksfunktion) *109*
fabsl (Bibliotheksfunktion) *109*
fail
ios-Elementfunktion *347*
pstream-Elementfunktion *373*
FAR-Heap
Speicher freigeben *110*
Speicher reservieren auf *111*
Speicherplatz reservieren auf *109*
Zeiger *111*
far-Zeiger *110*
Adreßsegment *126*
erzeugen *198*
Farben und Paletten
Hintergrundfarbe für Text *300*
Textattribute setzen *299*
Zeichenfarbe setzen *301*
farcalloc (Bibliotheksfunktion) *109*
farfree (Bibliotheksfunktion) *110*
farmalloc (Bibliotheksfunktion) *111*
farrealloc (Bibliotheksfunktion) *111*
FAT (File Allocation Table)
Informationen über *146, 147*
fclose (Bibliotheksfunktion) *112*
fcloseall (Bibliotheksfunktion) *112*
fcntl.h (Header-Datei) *26*
fcvt (Bibliotheksfunktion) *113*
fd
filebuf-Elementfunktion *341*
fdopen (Bibliotheksfunktion) *113*
Fehler
bei Berechnungen *189*
Bestimmung bei Streams *115*
DOS *88*
erweiterte Bestimmung *88*
Flag *70*
beim gemeinsamen Datenzugriff *180,181*
Konstanten und Texte *329*
Lesen/Schreiben *115*
mathematische *189*
Fehlermeldungen *279, 280*
_strerror *279*
strerror *280*
ausgeben *209, 328*
Fehlersuche
Routinen zur Diagnose *29*
Feldermeldungen
Zeiger auf *279, 280*
Fenster
Bildlauf ein-/ausschalten *335*
feof (Bibliotheksfunktion) *114*
ferror (Bibliotheksfunktion) *115*
ferror (Bibliotheksmakro) *115*
Festplatte
siehe auch Disketten *144*
fflush (Bibliotheksfunktion) *115*
fgetpos (Bibliotheksfunktion) *116*
fgets (Bibliotheksfunktion) *117*
filebuf (Klasse) *339*
filelength (Bibliotheksfunktion) *117*
fileno (Bibliotheksmakro) *118*
FileNull
TFile-Datentyp *516*
fill
ios-Elementfunktion *347*
find
ipstream-Elementfunktion *366*
String-Elementfunktion *522*
TBinarySearchTreeImp-Elementfunktion *408*
TIBinarySearchTreeImp-Element-
funktion *410*
TMArrayAsVector-Elementfunktion *386*
TMBagAsVector-Elementfunktion *404*
TMCVectorImp-Elementfunktion *476*
TMDictionaryAsHashTable-Element-
funktion *423*
TMHashTableImp-Elementfunktion *439*
TMIArrayAsVector-Elementfunktion *391*

TMICVectorImp-Elementfunktion *483*
TMIDictionaryAsHashTable-
Elementfunktion *426*
TMIHashTableImp-Elementfunktion *442*
find_first_not_of
string-Elementfunktion *522*
find_first_of
string-Elementfunktion *522*
find_last_not_of
string-Elementfunktion *523*
find_last_of
string-Elementfunktion *523*
FindBase
TStreamableBase-Elementfunktion *375*
FindDetach
TMDoubleListImp-Elementfunktion *431*
TMISListImp-Elementfunktion *451*
TMListImp-Elementfunktion *446*
TMSDoubleListImp-Elementfunktion *433*
findfirst (Bibliotheksfunktion) *118*
FindMember
TMBagAsVector-Elementfunktion *403*
TMIBagAsVector-Elementfunktion *406*
findnext (Bibliotheksfunktion) *120*
findObject
opstream-Elementfunktion *369*
FindPred
TMDoubleListImp-Elementfunktion *431*
TMIDoubleListImp-Elementfunktion *436*
TMIListImp-Elementfunktion *449*
TMISListImp-Elementfunktion *451*
TMListImp-Elementfunktion *446*
TMSDoubleListImp-Elementfunktion *433*
findVB
opstream-Elementfunktion *370*
FirstDayOfMonth
TDate-Elementfunktion *513*
FirstThat
TMArrayAsVector-Elementfunktion *385*
TMDequeAsDoubleList-Element-
funktion *418*
TMDequeAsVector-Elementfunktion *412*
TMDoubleListImp-Elementfunktion *430*
TMIArrayAsVector-Elementfunktion *390*
TMIBagAsVector-Elementfunktion *406*
TMIDequeAsDoubleList-Element-
funktion *421*
TMIDequeAsVector-Elementfunktion *416*
TMIDoubleListImp-Elementfunktion *435*
TMIListImp-Elementfunktion *449*
TMIQueueAsDoubleList-Element-
funktion *458*
TMIQueueAsVector-Elementfunktion *454*
TMIStackAsVector-Elementfunktion *466*
TMIVectorImp-Elementfunktion *480*
TMListImp-Elementfunktion *445*
TMQueueAsDoubleList-Element-
funktion *456*
TMQueueAsVector-Elementfunktion *452*
TMStackAsVector-Elementfunktion *464*
TMVectorImp-Elementfunktion *471*
Flags
carry *163, 164, 165, 166*
Control-Break *82*
Dateiende *70, 114*
Dateizugriff *265*
Fehler *70*
für Bildschirmausgabe *327*
für printf *218*
ios-Elementfunktion *347*
Lesen/Schreiben *96, 238*
Verify *259*
Verify (DOS) *153*
float.h (Header-Datei) *26*
floor (Bibliotheksfunktion) *121*
floorl (Bibliotheksfunktion) *121*
flush
opstream-Elementfunktion *370*
ostream-Elementfunktion *354*
TBinarySearchTreeImp-Elementfunktion *408*
TFile-Elementfunktion *518*
TIBinarySearchTreeImp-Element-
funktion *410*
TMArrayAsVector-Elementfunktion *385*
TMBagAsVector-Elementfunktion *403*
TMDequeAsDoubleList-Element-
funktion *418*
TMDequeAsVector-Elementfunktion *412*
TMDictionaryAsHashTable-Element-
funktion *423*
TMDoubleListImp-Elementfunktion *430*
TMHashTableImp-Elementfunktion *439*
TMIArrayAsVector-Elementfunktion *391*
TMIBagAsVector-Elementfunktion *406*
TMIDequeAsDoubleList-Element-
funktion *421*
TMIDequeAsVector-Elementfunktion *416*

TMIDictionaryAsHashTable-Elementfunktion *426*
TMIDoubleListImp-Elementfunktion *435*
TMIHashTableImp-Elementfunktion *442*
TMIQueueAsDoubleList-Elementfunktion *458*
TMIQueueAsVector-Elementfunktion *454*
TMIStackAsVector-Elementfunktion *466*
TMIVectorImp-Elementfunktion *480*
TMListImp-Elementfunktion *445*
TMQueueAsDoubleList-Elementfunktion *456*
TMQueueAsVector-Elementfunktion *452*
TMStackAsVector-Elementfunktion *464*
TMVectorImp-Elementfunktion *471*
flushall (Bibliotheksfunktion) *121*
fmod (Bibliotheksfunktion) *122*
fmodl (Bibliotheksfunktion) *122*
fname
Type_info-Elementfunktion *507*
fnmerge (Bibliotheksfunktion) *123*
fnsplit (Bibliotheksfunktion) *124*
fopen (Bibliotheksfunktion) *125*
ForEach
TBinarySearchTreeImp-Elementfunktion *408*
TIBinarySearchTreeImp-Elementfunktion *410*
TMArrayAsVector-Elementfunktion *385*
TMBagAsVector-Elementfunktion *403*
TMDequeAsDoubleList-Elementfunktion *418*
TMDequeAsVector-Elementfunktion *412*
TMDictionaryAsHashTable-Elementfunktion *423*
TMDoubleListImp-Elementfunktion *430*
TMHashTableImp-Elementfunktion *439*
TMIArrayAsVector-Elementfunktion *391*
TMIBagAsVector-Elementfunktion *406*
TMIDequeAsDoubleList-Elementfunktion *421*
TMIDequeAsVector-Elementfunktion *416*
TMIDictionaryAsHashTable-Elementfunktion *426*
TMIDoubleListImp-Elementfunktion *435*
TMIHashTableImp-Elementfunktion *442*
TMIListImp-Elementfunktion *449*
TMIQueueAsDoubleList-Elementfunktion *458*
TMIQueueAsVector-Elementfunktion *454*
TMIStackAsVector-Elementfunktion *466*
TMIVectorImp-Elementfunktion *480*
TMListImp-Elementfunktion *445*
TMQueueAsDoubleList-Elementfunktion *456*
TMQueueAsVector-Elementfunktion *452*
TMStackAsVector-Elementfunktion *464*
TMVectorImp-Elementfunktion *472*
Format-Spezifizierer
scanf und *240*
Formatieren
vsprinf *317*
vsscanf *317*
Formatierung
Ausgabe *76*
cprintf *76*
fprintf *127*
fscanf *131*
prinf *216*
sprintf *271*
vfprintf *314*
Formatierungsanweisungen
in scanf *241*
Formatierungsanweisungen (für printf) *216*
Formatstring *280*
in printf *216*
Formatstrings
Ausgabe *216*
FP_OFF (Bibliotheksmakro) *126*
FP_SEG (Bibliotheksmakro) *126*
fpbase (Klasse) *364*
fprintf (Bibliotheksfunktion) *127*
fputchar (Bibliotheksfunktion) *128, 223*
fputs (Bibliotheksfunktion) *128*
fread (Bibliotheksfunktion) *129*
freadBytes
ipstream-Elementfunktion *366*
freadString
ipstream-Elementfunktion *366*
free (Bibliotheksfunktion) *129*
freeze
strstreambuf-Elementfunktion *360*
freopen (Bibliotheksfunktion) *130*
frexp (Bibliotheksfunktion) *131*
frexpl (Bibliotheksfunktion) *131*
fscanf (Bibliotheksfunktion) *131*
fseek (Bibliotheksfunktion) *132*
fsetpos (Bibliotheksfunktion) *133*

fstat (Bibliotheksfunktion) *135*
fstream (Klasse) *342*
fstream.h (Header-Datei) *26*
fstreambase (Klasse) *343*
ftell (Bibliotheksfunktion) *137*
ftime (Bibliotheksfunktion) *137*
Funktionen
 80x86 *32*
 BIOS- *32*
 Datum und Zeit *36*
 des Betriebssystems *32*
 Ein-/Ausgabe *30*
 Ergebnis von main *40*
 main *37*
 mit mehreren Prototypen *28*
 String- *33*
 von der locale-Einstellung betroffene *32*
 zur Konvertierung *29*
 zur locale-Einstellung *34*
 zur Prozeßsteuerung *35*
 zur Speicherverwaltung *33, 34*
 zur Zeichenklassifizierung *28*
fwrite (Bibliotheksfunktion) *139*
fwritebytes
 opstream-Elementfunktion *370*
fwriteString
 opstream-Elementfunktion *370*

G

Gameport *57, 58*
gbump
 streambuf-Elementfunktion *358*
gcount
 istream-Elementfunktion *350*
gcvt (Bibliotheksfunktion) *139*
Genauigkeit
 in Gleitkommaoperationen *73*
generic.h (Header-Datei) *26*
geninterrupt (Bibliotheksmakro) *139*
Gerätetreiber *167*
get
 istream-Elementfunktion *350*
 TMIQueueAsDoubleList-Elementfunktion *458*
 TMIQueueAsVector-Elementfunktion *454*
 TMQueueAsDoubleList-Elementfunktion *456*
 TMQueueAsVector-Elementfunktion *453*
get_at
 string-Elementfunktion *523*
 TSubString -Elementfunktion *531*
get_case_sensitive_flag
 string-Elementfunktion *523*
get_initial_capacity
 string-Elementfunktion *523*
get_max_waste
 string-Elementfunktion *523*
get_paranoid_check
 string-Elementfunktion *524*
get_resize_increment
 string-Elementfunktion *524*
get_skipwhitespace_flag
 string-Elementfunktion *524*
getc (Bibliotheksmakros) *140*
getcbrk (Bibliotheksfunktion) *140*
getchar (Bibliotheksfunktion) *141*
getche (Bibliotheksfunktion *142*
getcurdir (Bibliotheksfunktion) *142*
getcwd (Bibliotheksfunktion) *142*
getdate (Bibliotheksfunktion) *91*
GetDelta
 TMCVectorImp-Elementfunktion *476, 480*
 TMVectorImp-Elementfunktion *472*
getdfree (Bibliotheksfunktion) *144*
getdisk (Bibliotheksfunktion) *145*
getdta (Bibliotheksfunktion) *145*
getenv (Bibliotheksfunktion) *146*
getfat (Bibliotheksfunktion) *146*
getfatd (Bibliotheksfunktion) *147*
getftime (Bibliotheksfunktion) *147*
GetHandle
 TFile-Elementfunktion *518*
GetItemsInContainer
 TBinarySearchTreeImp-Elementfunktion *408*
 TIBinarySearchTreeImp-Element-
funktion *410*
 TMArrayAsVector-Elementfunktion *385*
 TMBagAsVector-Elementfunktion *403*
 TMDequeAsDoubleList-Elementfunktion *419*
 TMDequeAsVector-Elementfunktion *412*
 TMDictionaryAsHashTable-Elementfunktion *423*

TMHashTableImp-Elementfunktion *439*
TMIArrayAsVector-Elementfunktion *391*
TMIBagAsVector-Elementfunktion *406*
TMIDequeAsDoubleList-Elementfunktion *421*
TMIDequeAsVector-Elementfunktion *416*
TMIDictionaryAsHashTable-Elementfunktion *426*
TMIDoubleListImp-Elementfunktion *435*
TMIHashTableImp-Elementfunktion *442*
TMIQueueAsDoubleList-Elementfunktion *458*
TMIQueueAsVector-Elementfunktion *455*
TMIStackAsVector-Elementfunktion *466*
TMQueueAsDoubleList-Elementfunktion *456*
TMStackAsVector-Elementfunktion *464*
GetLeft
TMDequeAsDoubleList-Elementfunktion *419*
TMDequeAsVector-Elementfunktion *412*
TMIDequeAsDoubleList-Element funktion *421*
TMIDequeAsVector-Elementfunktion *416*
getline
istream-Elementfunktion *351*
string-Elementfunktion *531*
GetObject
TStreamer-Elementfunktion *376*
getpass (Bibliotheksfunktion) *148*
getpid (Bibliotheksfunktion) *149*
GetPriority
TThread-Elementfunktion *539*
getpsp (Bibliotheksfunktion) *149*
GetRight
TMDequeAsDoubleList-Elementfunktion *419*
TMDequeAsVector-Elementfunktion *413*
TMIDequeAsDoubleList-Elementfunktion *421*
TMIDequeAsVector-Elementfunktion *416*
gets (Bibliotheksfunktion) *150*
GetStatus
TFile-Elementfunktion *518*
TThread-Elementfunktion *539*
gettext (Bibliotheksfunktion) *150*
gettextinfo (Bibliotheksfunktion) *151*
gettime (Bibliotheksfunktion) *152*
getvect (Bibliotheksfunktion) *152*
getverify (Bibliotheksfunktion) *153*
getVersion
ipstream-Elementfunktion *366*
Gleitkomma
Software-Signal *227*
Gleitkomma-Paket
clear87_ *69*
Steuerwort *73*
Gleitkommafunktionen *33*
Status-WORD *273*
Gleitkommapaket
Re-Initialisierung *126*
Gleitkommazahlen
absoluten Wert berechnen *109*
in Strings konvertieren *113*
Strings und *51*
globale Variablen *325*, *328*
siehe auch Variablen *325*
gmtime (Bibliotheksfunktion) *154*
good
ios-Elementfunktion *347*
pstream-Elementfunktion *373*
goto, nicht-lokales *183*
gotoxy
conbuf-Elementfunktion *338*
gotoxy (Bibliotheksfunktion) *155*
gptr
streambuf-Elementfunktion *358*
Grafiktreiber
Modus *150*
Greenwich Mean Time (GMT) *82*, *137*, *180*
konvertieren in *154*
setzen *307*
timezone (glob. Variable) *334*
Großbuchstaben
in Kleinbuchstaben kovertieren *305*, *306*
Konvertierung zu *306*, *307*
prüfen auf *173*
zu Kleinbuchtaben *283*
Größe
Datei *69*, *117*
Speicher *58*, *59*, *60*
Grow
TMArrayAsVector-Elementfunktion *386*
TMIArrayAsVector-Elementfunktion *392*

H

Handles

Datei- *86, 118*
Hardware
Ein-/Ausgabe *167*
Gerätetyp prüfen *170*
Interrupts aktivieren/deaktivieren *85*
Schnittstellen *161, 162, 207*
Überprüfung auf Interrupts *57*
Hash
String-Elementfunktion *524*
TDate-Elementfunktion *513*
TTime-Elementfunktion *542*
HashTable
TMDictionaryAsHashTable-Datenelement *423*
HashValue
TMDDAssociation-Elementfunktion *397*
TMDIAssociation-Elementfunktion *398*
TMIDAssociation-Elementfunktion *400*
TMIIAssociation-Elementfunktion *401*
HasMember
TMArrayAsVector-Elementfunktion *385*
TMBagAsVector-Elementfunktion *403*
TMIArrayAsVector-Elementfunktion *391*
TMIBagAsVector-Elementfunktion *406*
Head
TMDoubleListImp-Datenelement *431*
TMListImp-Datenelement *446*
Header-Dateien
alloc.h *26*
assert.h *26*
bcd.h *26*
bios.h *26*
checks.h *26*
complex.h *26*
conio.h *26*
constrea.h *26*
cstring.h *26*
ctype.h *26*
dir.h *26*
direct.h *26*
dirent.h *26*
dos.h *26*
errno.h *26*
except.h *26*
excpt.h *26*
fcntl.h *26*
float.h *26*
fstream.h *26*
generic.h *26*
io.h *26*
iomanip.h *27*
iostream.h *27*
limits.h *27*
Liste der *26*
locale.h *27*
malloc.h *27*
math.h *27*
mem.h *27*
new.h *27*
process.h *27*
ref.h *27*
regexp.h *27*
search.h *27*
setjmp.h *27*
share.h *27*
signal.h *27*
stdarg.h *27*
stddef.h *27*
stdio.h *27*
stdiostr.h *27*
stdlib.h *27*
string.h *27*
strstrea.h *27*
sys\locking.h *27*
sys\stat.h *27*
sys\timeb.h *27*
sys\types.h *27*
time.h *27*
typeinfo.h *28*
utime.h *28*
values.h *28*
varargs.h *28*
Heap
erweitern *156*
FAR *109, 110, 111*
freie Blöcke füllen *158, 159*
Knoten überprüfen *157*
siehe auch FAR-Heap, NEAR-Heap *109*
Speicher reservieren *188*
überprüfen *156, 157, 159, 236*
unbenutzte Blöcke freigeben *158*
Verwaltung *64*
heapcheck (Bibliotheksfunktion) *156*
heapcheckfree (Bibliotheksfunktion) *156*
heapchecknode (Bibliotheksfunktion) *157*
heapfillfree (Bibliotheksfunktion) *158*
heapwalk (Bibliotheksfunktion) *159*
hexadezimale Zahlen

prüfen auf *174*
Hierarchie
Streams *363*
highvideo
conbuf-Elementfunktion *338*
highvideo (Bibliotheksfunktion) *160*
Hintergrundfarbe
für Textausgaben *300*
Hohe Intensität *160*
Hour
TTime-Elementfunktion *542*
HourGMT
TTime-Elementfunktion *542*
HowToPrint
TDate-Datentyp *511*
hyperbolische Funktionen *74, 264, 297*
hyperbolischer Cosinus *74*
hypot (Bibliotheksfunktion) *160*
Hypotenuse
berechnen *160*
hypotl (Bibliotheksfunktion) *160*

I

I/O-Funktionen
setftime *147*
I/O-Routinen
_dos_creat *86*
_dos_creatnew *87*
_dos_open *95*
_dos_read *97*
_dos_write *99*
_fsopen *133*
_get_osfhandle *149*
_open *203*
_rtl_chmod *234*
_rtl_close *235*
_rtl_creat *236*
_rtl_open *237*
_rtl_read *239*
_rtl_write *239*
access *46*
cgets *66*
chmod *68*
clearerr *70*
close *71*
clreol *72*
clrscr *72*
cprintf *76*
creat *77*
creatnew *78*
creattemp *79*
cscanf *80*
delline *84*
dup *100*
dup2 *101*
eof *104*
fclose *112*
fcloseall *112*
feof *114*
ferror *115*
fflush *115*
fgetc *116*
fgetchar *116*
fgets *117*
filelength *117*
fileno *118*
flushall *121*
fopen *125*
fprintf *127*
fputc *128*
fputchar *128, 223*
fputs *128*
fread *129*
freopen *130*
fscanf *131*
fseek *132*
fstat *135*
ftell *137*
fwrite *139*
getc *140*
getch *141*
getche *142*
getpass *148*
gets *150*
ioctl *167*
isatty *170*
kbhit *175*
lock *180*
locking *181*
lseek *186*
open *203*
perror *209*
printf *216*
putc *222*
putch *222*
puts *224*

putw 225
read 229
remove 231
rename 232
rewind 232
scanf 240
setbuf 249
setmode 257
setvbuf 258
sopen 265
sprintf 271
sscanf 272
stat 135
tell 297
umask 309
unlock 311
vfprintf 314
vfscanf 314
vprintf 315
vscanf 316
vsprintf 317
vsscanf 317
write 322
ifpstream (Klasse) 364
ifstream (Klasse) 344
ignore
istream-Elementfunktion 351
imag
complex-Elementfunktion 494
IMPLEMENT_CASTABLE (Makros) 382
IMPLEMENT_CASTABLE_ID (Makro) 381
IMPLEMENT_STREAMABLE (Makros) 380
IMPLEMENT_STREAMABLE_CLASS
(Makro) 381
IMPLEMENT_STREAMABLE_CTOR
(Makros) 381
IMPLEMENT_STREAMABLE_FROM_BASE
(Makro) 382
IMPLEMENT_STREAMABLE_POINTER
(Makro) 381
IMPLEMENT_STREAMER (Makro) 382
in_avail, streambuf-Elementfunktion 356
IndexOfMonth
TDate-Elementfunktion 513
init
ios-Elementfunktion 349
pstream-Elementfunktion 374
initial_capacity
String-Elementfunktion 524
Initialisierung
des Zufallszahlen-Generators 272
des Zufallszahlengenerators 228
Speicher 197, 257
inp (Bibliotheksmakro) 161
inport (Bibliotheksfunktion) 162
inportb (Bibliotheksmakro) 162
inpw (Bibliotheksmakro) 162
insert
string-Elementfunktion 524
InsertEntry
TMArrayAsVector-Elementfunktion 386
TMIArrayAsVector-Elementfunktion 392
insline
conbuf-Elementfunktion 338
insline (Bibliotheksfunktion) 163
int (Datentyp)
Konvertierung in String 308
Vertauschung der Worthälften 295
int86 (Bibliotheksfunktion) 163
int86x (Bibliotheksfunktion) 164
intdos (Bibliotheksfunktion) 165
intdosx (Bibliotheksfunktion) 165
Integer
Absolutwert 45
aus Stream lesen 154
bitweise rotieren 234
dividieren 176
Konvertierung zu Strings 174
long 175
Schreiben in Streams 225
speichern 212
Intensität
hohe 160
niedrige 184
normal 202
Interpunktionszeichen
prüfen auf 172
Interrupt-Handler 66
DOS 82
Signal-Handler und 261
Interrupt-Vektoren 82
Lesen 95, 152
setzen 98
setzen und lesen 152
Interrupts
Control-Break 82
erzeugen 139
Hardware) 85

NMI (nicht maskierbar) *85*
siehe auch DOS-Interrupts *82*
Software *139*
Software-Interrupts auslösen *163, 164, 166*
Software-Signal *227*
Strg-Untbr *140*
intr (Bibliotheksfunktion) *166*
io.h (Header-Datei) *26*
ioctl (Bibliotheksfunktion) *167*
IOFBF_, IOLBF_, IONBF_ (Konstanten) *258*
iomanip.h (Header-Datei) *27*
ios (Klasse) *345*
iostream (Klasse) *349*
iostream.h (Header-Datei) *27*
iostream_withassign (Klasse) *349*
ipfx
istream-Elementfunktion *351*
ipstream (Klasse) *365*
is_null
string-Elementfunktion *524*
TSubString -Elementfunktion *531*
is_rtl_open, filebuf-Elementfunktion *341*
isalnum (Bibliotheksmakro) *169*
isalpha (Bibliotheksmakro) *169*
isascii (Bibliotheksmakro) *169*
isatty (Bibliotheksfunktion) *170*
iscntrl (Bibliotheksmakro) *170*
isdigit (Bibliotheksmakro) *171*
IsDST
TTime-Elementfunktion *543*
IsEmpty
TMArrayAsVector-Elementfunktion *386*
TMBagAsVector-Elementfunktion *403*
TMDequeAsDoubleList-Elementfunktion *419*
TMDequeAsVector-Elementfunktion *413*
TMDictionaryAsHashTable-Elementfunktion *424*
TMDoubleListImp-Elementfunktion *430*
TMHashTableImp-Elementfunktion *439*
TMIArrayAsVector-Elementfunktion *391*
TMIBagAsVector-Elementfunktion *406*
TMIDequeAsDoubleList-Elementfunktion *421*
TMIDequeAsVector-Elementfunktion *416*
TMIDictionaryAsHashTable-Elementfunktion *426*
TMIDoubleListImp-Elementfunktion *435*
TMIHashTableImp-Elementfunktion *442*
TMIQueueAsDoubleList-Elementfunktion *458*
TMIQueueAsVector-Elementfunktion *455*
TMIStackAsVector-Elementfunktion *467*
TMListImp-Elementfunktion *446*
TMQueueAsDoubleList-Elementfunktion *456*
TMQueueAsVector-Elementfunktion *453*
TMStackAsVector-Elementfunktion *464*
IsFull
TMArrayAsVector-Elementfunktion *386*
TMBagAsVector-Elementfunktion *404*
TMDequeAsDoubleList-Elementfunktion *419*
TMDequeAsVector-Elementfunktion *413*
TMIArrayAsVector-Elementfunktion *391*
TMIBagAsVector-Elementfunktion *406*
TMIDequeAsDoubleList-Elementfunktion *421*
TMIDequeAsVector-Elementfunktion *416*
TMIQueueAsDoubleList-Elementfunktion *458*
TMIQueueAsVector-Elementfunktion *455*
TMIStackAsVector-Elementfunktion *467*
TMQueueAsDoubleList-Elementfunktion *457*
TMQueueAsVector-Elementfunktion *453*
TMStackAsVector-Elementfunktion *464*
isgraph (Bibliotheksmakro) *171*
islower (Bibliotheksmakro) *171*
IsOpen
TFile-Elementfunktion *518*
isprint (Bibliotheksmakro) *172*
ispunct (Bibliotheksmakro) *172*
isspace (Bibliotheksmakro) *173*
istream (Klasse) *350*
istream_withassign (Klasse) *352*
istrstream (Klasse) *352*
isupper (Bibliotheksmakro) *173*
IsValid
TDate-Elementfunktion *513*
TTime-Elementfunktion *543*
isxdigit (Bibliotheksmakro) *174*
ItemAt
TMArrayAsVector-Elementfunktion *386*
TMIArrayAsVector-Elementfunktion *392*
IterFunc
TMArrayAsVector-Funktionstyp *384*
TMBagAsVector-Funktionstyp *402*

TMDequeAsDoubleList-Funktionstyp *418*
TMDequeAsVector-Funktionstyp *412*
TMDoubleListImp-Funktionstyp *429*
TMIArrayAsVector-Funktionstyp *389*
TMIBagAsVector-Funktionstyp *405*
TMIDequeAsDoubleList-Funktionstyp *420*
TMIDequeAsVector-Funktionstyp *415*
TMIDoubleListImp-Funktionstyp *434*
TMIListImp-Funktionstyp *449*
TMIStackAsVector-Funktionstyp *466*
TMIVectorImp-Funktionstyp *479*
TMListImp-Funktionstyp *445*
TMStackAsVector-Funktionstyp *463*
TMVectorImp-Funktionstyp *471*
itoa (Bibliotheksfunktion) *174*

J

Japanisches Datumsformat *76*
JDay
TDate-Elementfunktion *513*
Jokerzeichen
in Kommandozeilen *39*
JulTy
TDate-Datentyp *512*

K

Kalenderformat *199*
kbhit (Bibliotheksfunktion) *175*
Key
TMDDAssociation-Elementfunktion *397*
TMDIAssociation-Elementfunktion *398*
TMIDAssociation-Elementfunktion *400*
TMIIAssociation-Elementfunktion *401*
Kindprozeß *105*
warten auf *82*
Kindprozesse *104*, *267*
Funktionen zur Erzeugung *35*
Übergabe von Informationen *41*
Klassifizierungsroutinen
isalnum *169*
isalpha *169*
isascii *169*
iscntrl *170*
isdigit *171*
isgraph *171*
islower *171*
isprint *172*
ispunct *172*
isspace *173*
isupper *173*
isxdigit *174*
Kleinbuchstaben
konvertieren zu Großbuchstaben *294*, *306*, *307*
Konvertierung zu *305*, *306*
prüfen auf *171*
Kommandozeilen-Parameter
Beispielprogramm *38*
und Jokerzeichen *39*
Kommandozeilencompiler
-p (Pascal-Aufrufkonvention) *40*
Kommandozeilenparameter *326*
Komplexe Zahlen *492*
Absolutwert berechnen *63*
Arcussinus von *493*
Arcustangens von *493*
Cosinus hyperbolicus von *493*
Cosinus von *493*
Exponentialfunktion für *494*
Imaginärteil von *494*
konjugierter Wert von *493*
Logarithmus von *494*
natürlicher Logrithmus von *494*
polar-Funktion *494*
pow-Funktion *494*
Sinus hypberbolicus von *495*
Sinus von *495*
Tangens hyberbolicus von *495*
Tangens von *495*
und andere numerische Typen *492*
Wurzelfunktion von *495*
Konstanten
CLK_TCK *70*
FA_ARCH...FA_SYSTEM *119*
IOFBF_, IOLBF_ und IONBF *258*
P_WAIT, P_NOWAIT, P_OVERLAY *267*
S_IWRITE, S_IREAD *68*
SEEK_CUR, SEEK_END und SEEK_SET *132*
Konstruktoren
bcd *490*
complex *492*
conbuf *337*
constream *339*
DEFAULT_SET_SIZE) *460*, *461*, *462*
filebuf *340*

fpbase *364*
fstream *342*
fstreambase *343*
ifpstream *365*
ifstream *344*
ios *346*
iostream *349*
iostream_withassign *349*
ipstream *365, 367*
istream *350*
istream_withassign *352*
istrstream *352*
Lock *534*
ofpstream *368*
ofstream *353*
opstream *369, 371*
ostream *354*
ostream_withassign *355*
ostrstream *355*
pstream *372, 374*
streambuf *356*
string *519*
strstream *361*
strstreambase *359*
strstreambuf *359*
TArrayAsVector *389*
TArrayAsVectorIterator *389*
TBagAsVector *404*
TBagAsVectorIterator *405*
TBinarySearchTreeIteratorImp *409*
TCriticalSection *533*
TCVectorIteratorImp *477*
TDate *512*
TDDAssociation *398*
TDequeAsDoubleListIterator *420*
TDequeAsVector *415*
TDequeAsVectorIterator *415*
TDIAssociation *399*
TDictionaryAsHashTable *425*
TDictionaryAsHashTableIterator *425*
TDictionaryIterator *428*
TDoubleListImp *432*
TDoubleListIteratorImp *433*
TFile *517*
THashTableImp *441*
THashTableIteratorImp *441*
TIArrayAsVector *394*
TIArrayAsVectorIterator *394*
TIBagAsVector *407*
TIBagAsVectorIterator *407*
TIBinarySearchTreeIteratorImp *411*
TICVectorImp *484*
TICVectorIteratorImp *484*
TIDAssociation *400*
TIDequeAsDoubleListIterator *422*
TIDequeAsVector *417*
TIDequeAsVectorIterator *418*
TIDictionaryAsHashTable *427*
TIDictionaryAsHashTableIterator *428*
TIDoubleListIteratorImp *437*
TIHashTableImp *443*
TIHashTableIteratorImp *443*
TIIAssociation *402*
TIListIteratorImp *451*
TIQueueAsDoubleListIterator *459*
TIQueueAsVector *455*
TIQueueAsVectorIterator *456*
TISArrayAsVector *396*
TISArrayAsVectorIterator *396*
TISDoubleListIteratorImp *438*
TISetAsVectorIterator *463*
TISListIteratorImp *452*
TIStackAsListIterator *470*
TIStackAsVector *468*
TISVectorImp *485*
TISVectorIteratorImp *486*
TIVectorImp *482*
TIVectorIteratorImp *482*
TListIteratorImp *447*
TMArrayAsVector *384*
TMArrayAsVectorIterator *388*
TMBagAsVector *402*
TMBagAsVectorIterator *404*
TMCVectorImp *475, 477*
TMCVectorIteratorImp *476*
TMDDAssociation *397*
TMDequeAsDoubleListIterator *420*
TMDequeAsVector *412*
TMDequeAsVectorIterator *414*
TMDIAssociation *398*
TMDictionaryAsHashTable *423*
TMDictionaryAsHashTableIterator *424*
TMDoubleListElement *429*
TMDoubleListImp *430*
TMDoubleListIteratorImp *431*
TMHashTableImp *438*
TMHashTableIteratorImp *440*
TMIArrayAsVector *389*

TMIArrayAsVectorIterator *393*
TMIBagAsVector *405*
TMIBagAsVectorIterator *407*
TMICVectorImp *483*
TMICVectorIteratorImp *483*
TMIDAssociation *399*
TMIDequeAsDoubleListIterator *422*
TMIDequeAsVector *416*
TMIDequeAsVectorIterator *417*
TMIDictionaryAsHashTable *425*
TMIDictionaryAsHashTableIterator *426*
TMIDoubleListIteratorImp *436*
TMIHashTableImp *441*
TMIHashTableIteratorImp *442*
TMIIAssociation *401*
TMIListIteratorImp *450*
TMIQueueAsDoubleListIterator *459*
TMIQueueAsVector *454*
TMIQueueAsVectorIterator *455*
TMISArrayAsVector *396*
TMISDoubleListIteratorImp *438*
TMISetAsVectorIterator *462*
TMISListIteratorImp *451*
TMIStackAsListIterator *469*
TMIStackAsVector *466*
TMIStackAsVectorIterator *467, 468*
TMISVectorImp *485*
TMISVectorIteratorImp *485*
TMIVectorImp *480*
TMIVectorIteratorImp *481*
TMListElement *444*
TMListImp *445*
TMListIteratorImp *446*
TMQueueAsDoubleListIterator *457*
TMQueueAsVector *452*
TMQueueAsVectorIterator *453*
TMSArrayAsVector *394*
TMSArrayAsVectorIterator *395*
TMSDoubleListIteratorImp *433*
TMSetAsVectorIterator *460*
TMSListIteratorImp *448*
TMStackAsListIterator *468*
TMStackAsVector *463*
TMStackAsVectorIterator *465*
TMSVectorImp *478*
TMSVectorIteratorImp *478*
TMutex *535*
TMutex::Lock *535*
TMVectorImp *471*
TMVectorIteratorImp *473*
TQueueAsDoubleListIterator *457*
TQueueAsVector *453*
TQueueAsVectorIterator *454*
TSArrayAsVector *395*
TSArrayAsVectorIterator *395*
TSDoubleListIteratorImp *434*
TSetAsVectorIterator *461*
TShouldDelete *486*
TStackAsListIterator *469*
TStackAsVector *465*
TStackAsVectorIterator *465*
TStreamableClass *375*
TStreamer *377*
TSVectorImp *479*
TSVectorIteratorImp *479*
TSync *536*
Tsync::Lock *537*
TThread *538*
TTime *542*
TVectorImp *474*
TVectorIteratorImp *475*
xalloc *508*
xmsg *508*
Konvertierungen
BCD in long double *492*
Funktionen für *29*
komplexer Zahlen *495*
Konvertierungs-Routinen
_tolower *305*
_toupper *306*
Datum und Uhrzeit *99, 179, 199*
ltoa *187*
toascii *305*
tolower *306*
toupper *307*
ultoa *308*
Konvertierungsroutinen
_atold *51*
atof *51*
atoi *52*
atol *53*
Datum und Uhrzeit *48, 81, 154*
ecvt *101*
fcvt *113*
gcvt *139*
Gleitkommazahl zu String *101*
itoa *174*
numerisch nach String *174*

strtod *290*
strtol *293*

Koordinaten
Cursorposition *155, 320, 321*
Kopieren
Daten *200*
Koprozessor
siehe mathematischer Koprozessor *325*

L

labs (Bibliotheksfunktion) *175*
länderspezifische Einstellung *32, 34*
landesspezifische Formate *75*
Länge
von Dateien *69, 117*
von Strings *283*
LastThat
TMArrayAsVector-Elementfunktion *386*
TMDequeAsDoubleList-Elementfunktion *419*
TMDequeAsVector-Elementfunktion *413*
TMDoubleListImp-Elementfunktion *430*
TMIArrayAsVector-Elementfunktion *391*
TMIBagAsVector-Elementfunktion *406*
TMIDequeAsDoubleList-Elementfunktion *421*
TMIDequeAsVector-Elementfunktion *416*
TMIDoubleListImp-Elementfunktion *435*
TMIListImp-Elementfunktion *449*
TMIQueueAsDoubleList-Element funktion *458*
TMIQueueAsVector-Elementfunktion *455*
TMIStackAsVector-Elementfunktion *467*
TMIVectorImp-Elementfunktion *480*
TMListImp-Elementfunktion *446*
TMQueueAsDoubleList-Elementfunktion *457*
TMQueueAsVector-Elementfunktion *453*
TMStackAsVector-Elementfunktion *464*
TMVectorImp-Elementfunktion *472*
Laufwerk
aktuelle Nummer *92*
setzen *68*
Überprüfung auf Vorhandensein *57*
Laufwerke
aktuelles Verzeichnis *142*
Nummer *145*
Laufzeitbibltiothek
Funktionen der *28*
ldexp (Bibliotheksfunktion) *175*
ldexpl (Bibliotheksfunktion) *175*
ldiv (Bibliotheksfunktion) *176*
Leap
TDate-Elementfunktion *513*
Leerzeichen
prüfen auf *173*
Left
TMDequeAsVector-Datenelement *413*
Length
String-Elementfunktion *524*
TFile-Elementfunktion *518*
TSubString -Elementfunktion *531*
Lese-/Schreibzugriff *309*
Lesefehler *115*
Lesen
von der Tastatur *66*
Lesen-/Schreiben-Zugriff
prüfen auf *46*
LessThan
TBinarySearchTreeImp-Elementfunktion *408*
TIBinarySearchTreeImp-Elementfunktion *410*
lfind (Bibliotheksfunktion) *176*
Lim
TMVectorImp-Datenelement *473*
Limit
TMIVectorImp-Elementfunktion *480*
TMVectorImp-Elementfunktion *472*
limits.h (Header-Datei) *27*
lineare Suche *176*
lineare Suchen *185*
locale-Einstellung *32, 34*
auswählen *253*
locale.h (Header-Datei) *27*
localeconv (Bibliotheksfunktion) *177*
localtime (Bibliotheksfunktion) *179*
lock (Bibliotheksfunktion) *180*
locking (Bibliotheksfunktion) *181*
LockRange
TFile-Elementfunktion *518*
log
complex-Elementfunktion *494*
log (Bibliotheksfunktion) *182*
log10
complex-Elementfunktion *494*

log10 (Bibliotheksfunktion) *183*
log10l (Bibliotheksfunktion) *183*
Logarithmus
natürlicher *182*
zur Basis 26 *183*
logl (Bibliotheksfunktion) *182*
lokale Struktur
Zeiger auf *177*
long-Wert
zu String konvertieren *187*
longjmp (Bibliotheksfunktion) *183*
Löschen
Bildschirm im Textmodus *72*
bis zum Zeilenende *72*
Dateien *231, 311*
Verzeichnis *233*
von Textzeilen *84*
Zeile in Textfenster *72*
LowerBound
TMArrayAsVector-Elementfunktion *386*
TMIArrayAsVector-Elementfunktion *392*
lowvideo
conbuf-Elementfunktion *338*
lowvideo (Bibliotheksfunktion) *184*
lsearch (Bibliotheksfunktion) *185*
lseek (Bibliotheksfunktion) *186*
ltoa (Bibliotheksfunktion) *187*

M

main
Argumente *326*
main-Funktion *37*
als C-Routine deklarieren *40*
Beispiel *38*
Ergebnis *40*
Jokerzeichen *39*
Parameter *37*
Makro
peek *209*
peekb *209*
poke *212*
pokeb *213*
putc *222*
putchar *223*
Makros
_tolower *305*
_toupper *306*
abs *45*
assert *49*
Ausgabe zu Schnittstellen *207*
far-Zeiger erstellen *198*
ferror *115*
fileno *118*
FP_OFF *126*
FP_SEG *126*
geninterrupt *139*
getc *140*
getchar *141*
isalnum *169*
isalpha *169*
isascii *169*
iscntrl *170*
isdigit *171*
isgraph *171*
islower *171*
isprint *172*
ispunct *172*
isspace *173*
isupper *173*
isxdigit *174*
max *191*
min *197*
MK_FP *198*
outp *206*
peek *209*
peekb *209*
poke *212*
pokeb *212*
putc *222*
random *228*
randomize *228*
remove *231*
sopen *265*
strcmpi *276*
strncmpi *285*
toascii *305*
va_arg *313*
va_end *313*
va_start *313*
Vergleich zweier Werte *191, 197*
zu Diagnosezwecken *497, 499, 501*
malloc (Bibliotheksfunktion) *188*
malloc.h (Header-Datei) *27*
Mantisse *200*
einer Gleitkommazahl *131*
Maschinensprache

Literale in Objektcode einfügen *102*
math.h (Header-Datei) *27*
mathematische Routinen
_atold *51*
_fpreset *126*
_matherrl *189*
_status87 *273*
abs *45*
acos *46*
asin *48*
asinl *48*
atan *49*
atan2 *50*
atan2l *50*
atanl *49*
atof *51*
atoi *52*
atol *53*
cabs *63*
cabsl *63*
ceil *64*
ceill *64*
clear87_ *69*
control87_ *73*
cos *74*
cosh *74*
coshl *74*
cosl *74*
ecvt *101*
exp *108*
expl *108*
fabs *109*
fabsl *109*
fcvt *113*
floor *121*
floorl *121*
fmod *122*
fmodl *122*
frexp *131*
frexpl *131*
gcvt *139*
hyperbolische Funktionen *74, 264, 297*
hypot *160*
hypotl *160*
itoa *174*
labs *175*
ldexp *175*
ldexpl *175*
log *182*
log10 *183*
log10l *183*
logl *182*
ltoa *187*
matherr *189*
modf *200*
modfl *200*
poly *213*
polyl *213*
pow *214*
pow10 *215*
pow10l *215*
powl *214*
rand *227*
sin *263*
sinh *264*
sinhl *264*
sinl *263*
sqrt *271*
sqrtl *271*
srand *272*
strtod *290*
strtol *293*
tan *296*
tanh *297*
tanhl *297*
tanl *296*
ultoa *308*
Mathematischer Koprozessor *126, 325*
Ausnahmezustände *70*
Statuswort löschen *69*
Steuerwort *73*
Überprüfung auf Vorhandensein *58, 59*
matherr (Bibliotheksfunktion) *189*
Max
TDate-Elementfunktion *513*
TTime-Elementfunktion *543*
max (Makro) *191*
max_waste
String-Elementfunktion *524*
MaxDate
TTime-Datenelement *541*
mblen (Bibliotheksfunktion) *192*
mbstowcs (Bibliotheksfunktion) *192*
mbtowc (Bibliotheksfunktion) *193*
mem.h (Header-Datei) *27*
memccpy (Bibliotheksfunktion) *194*
memchr (Bibliotheksfunktion) *194*
memcmp (Bibliotheksfunktion) *195*

memcpy (Bibliotheksfunktion) *195*
memicmp (Bibliotheksfunktion) *196*
memmove (Bibliotheksfunktion) *196*
memset (Bibliotheksfunktion) *197*
mexcep_ (Datentyp) *190*
MEZ *82*
Min
 TDate-Elementfunktion *513*
 TTime-Elementfunktion *543*
min (Makro) *197*
Minute
 TTime-Elementfunktion *543*
MinuteGMT
 TTime-Elementfunktion *543*
MK_FP (Bibliotheksmakros) *198*
mkdir (Bibliotheksfunktion) *198*
mktemp (Bibliotheksfunktion) *199*
mktime (Bibliotheksfunktion) *199*
modf (Bibliotheksfunktion) *200*
modfl (Bibliotheksfunktion) *200*
Modulo *122*
Modus
 Datei *114, 130*
 Datei- *87, 93*
 Dateien *77, 78, 125*
Month
 TDate-Datentyp *512*
 TDate-Elementfunktion *514*
Monthname

 TDate-Elementfunktion *514*
MostDerived
 TStreamableBase-Elementfunktion *375*
movedata (Bibliotheksfunktion) *200*
movetext (Bibliotheksfunktion) *201*
movmem (Bibliotheksfunktion) *201*

N

name
 Type_info-Elementfunktion *507*
NameOfDay
 TDate-Elementfunktion *514*
NameOfMonth
 TDate-Elementfunktion *514*
NaN (Not a Number)
 und atof *52*
nationale Formate *75*
natürlicher Logarithmus *182*
NDEBUG-Symbol *49*
neue Dateien *77, 78, 86, 87, 236*
new.h (Header-Datei) *27*
Next
 TMDequeAsVector-Elementfunktion *414*
 TMDoubleListElement-Datenelement *429*
 TMListElement-Datenelement *444*
nicht-lokales goto *183*
niedrige Intensität *184*
norm
 complex-Elementfunktion *494*
normale Intensität *202*
normvideo
 conbuf-Elementfunktion *338*
normvideo (Bibliotheksfunktion) *202*
numerischer Koprozessor *273*

O

Objektcode
 Literale im *102*
oem_to_ansi
 String-Elementfunktion *524*
Öffnen
 von Dateien *265*
Offset
 far-Zeiger *198*
 von far-Adressen *126*
offsetof (Bibliotheksmakro) *202*
ofpstream (Klasse) *368*
ofstream (Klasse) *353*
open
 filebuf-Elementfunktion *341*
 fpbase-Elementfunktion *364*
 fstream-Elementfunktion *343*
 fstreambase-Elementfunktion *344*
 ifpstream-Elementfunktion *365*
 ifstream-Elementfunktion *345*
 ofpstream-Elementfunktion *369*
 ofstream-Elementfunktion *353*
 TFile-Elementfunktion *518*
open (Bibliotheksfunktion) *203*
opendir (Bibliotheksfunktion) *205*
openprot
 filebuf-Datenelement *340*
operator !
 TSubString -Elementfunktion *533*

operator !(), pstream-Elementfunktion *373*
Operator !=
 string-Elementfunktion *529*
 TDate-Elementfunktion *515*
 TSubString -Elementfunktion *532*
 TTime-Elementfunktion *544*
 Type_info-Elementfunktion *507*
operator
 string-Elementfunktion *528*
 TMArrayAsVector-Elementfunktion *387*
 TMIArrayAsVector-Elementfunktion *393*
 TMIVectorImp-Elementfunktion *481*
 TMVectorImp-Elementfunktion *472*
 TSubString -Elementfunktion *533*
operator ()
 string-Elementfunktion *528*
 TSubString -Elementfunktion *533*
operator +
 string-Elementfunktion *528*, *531*
 TDate-Elementfunktion *515*
 TTime-Elementfunktion *544*
operator ++
 TBinarySearchTreeIteratorImp-Elementfunktion *409*
 TDate-Elementfunktion *515*
 TIBinarySearchTreeIteratorImp-Elementfunktion *411*
 TMArrayAsVectorIterator-Elementfunktion *388*
 TMDequeAsVectorIterator-Elementfunktion *414*
 TMDictionaryAsHashTableIterator-Elementfunktion *424*
 TMDoubleListIteratorImp-Elementfunktion *432*
 TMHashTableIteratorImp-Elementfunktion *440*
 TMIArrayAsVectorIterator-Elementfunktion *393*
 TMIDoubleListIteratorImp-Elementfunktion *436*
 TMIHashTableIteratorImp-Elementfunktion *443*
 TMIListIteratorImp-Elementfunktion *450*
 TMIVectorIteratorImp-Elementfunktion *481*
 TMListIteratorImp-Elementfunktion *447*
 TMVectorIteratorImp-Elementfunktion *474*
 TTime-Elementfunktion *544*
operator +=
 string-Elementfunktion *528*
 TTime-Elementfunktion *544*
operator -
 TDate-Elementfunktion *515*
 TTime-Elementfunktion *544*
operator --
 TDate-Elementfunktion *515*
 TMDoubleListIteratorImp-Elementfunktion *432*
 TTime-Elementfunktion *544*
operator -=
 TTime-Elementfunktion *544*
operator
 string-Elementfunktion *529*
 TDate-Elementfunktion *514*
 TTime-Elementfunktion *543*
operator <
 string-Elementfunktion *530*
 TTime-Elementfunktion *544*
operator <, opstream-Elementfunktion *372*
operator
 string-Elementfunktion *530*
 TDate-Elementfunktion *515*
 TTime-Elementfunktion *543*
operator =
 string-Elementfunktion *528*
 TMVectorImp-Elementfunktion *472*
 TSubString -Elementfunktion *532*
 TSync -Elementfunktion *536*
 TThread-Elementfunktion *540*
operator ==
 string-Elementfunktion *529*
 TDate-Elementfunktion *515*
 TMDIAssociation-Elementfunktion *399*
 TMIDAssociation-Elementfunktion *400*
 TMIIAssociation-Elementfunktion *401*
 TSubString -Elementfunktion *532*
 TTime-Elementfunktion *544*
 Type_info-Elementfunktion *507*
operator
 string-Elementfunktion *530*
 TDate-Elementfunktion *515*
 TTime-Elementfunktion *544*
operator =
 string-Elementfunktion *530*
 TDate-Elementfunktion *515*
 TTime-Elementfunktion *544*
operator >
 ipstream-Elementfunktion *368*

string-Elementfunktion *530*
TTime-Elementfunktion *544*
operator delete
TMDoubleListElement-Elementfunktion *429*
TMListElement-Elementfunktion *444*
operator int
TBinarySearchTreeIteratorImp-Elementfunktion *409*
TIBinarySearchTreeIteratorImp-Elementfunktion *411*
TMArrayAsVectorIterator-Elementfunktion *388*
TMDequeAsVectorIterator-Elementfunktion *414*
TMDictionaryAsHashTableIterator-Elementfunktion *424*
TMDoubleListIteratorImp-Elementfunktion *432*
TMHashTableIteratorImp-Elementfunktion *440*
TMIDictionaryAsHashTableIterator-Elementfunktion *427*
TMIHashTableIteratorImp-Elementfunktion *443*
TMIVectorIteratorImp-Elementfunktion *482*
TMListIteratorImp-Elementfunktion *447*
TMVectorIteratorImp-Elementfunktion *474*
operator new
TMDoubleListElement-Elementfunktion *429*
TMListElement-Elementfunktion *444*
operator void *()
pstream-Elementfunktion *373*
opfx
ostream-Elementfunktion *354*
opstream (Klasse) *369*
osfx
ostream-Elementfunktion *354*
ostream (Klasse) *354*
ostream_withassign (Klasse) *355*
ostrstream (Klasse) *355*
out_waiting
streambuf-Elementfunktion *356*
outp (Bibliotheksmakro) *206*
outport (Bibliotheksfunktion) *207*
outportb (Bibliotheksfunktion) *207*
outpw (Bibliotheksfunktion) *207*
overflow
conbuf-Elementfunktion *338*
filebuf-Elementfunktion *341*
strstreambuf-Elementfunktion *360*
Overlays
Coroutinen und *183*
OwnsElements
TShouldDelete-Elementfunktion *487*

P

-p (Pascal-Aufrufkonvention) *40*
P_WAIT, P_NOWAIT, P_OVERLAY (Konstanten) *267*
Paletten
siehe Farben und Paletten *77*
Parameter
für main *37*
Kommandozeile *326*
variable Anzahl *313*
parent process *105*
parent-Prozess *267*
Parent::IsEmpty
TBinarySearchTreeImp-Elementfunktion *408*
TIBinarySearchTreeImp-Elementfunktion *410*
parsfnm (Bibliotheksfunktion) *208*
Pascal-Aufrufkonventionen (-p) *40*
Paßwörter *148*
PATH
Environment-Variable *268*
PATH (DOS-Environment-Variable *104*
Pause
Programmausführung *264*
pbase
streambuf-Elementfunktion *358*
pbump
streambuf-Elementfunktion *358*
pcount
ostrstream-Elementfunktion *356*
peek
istream-Elementfunktion *351*
peek (Bibliotheksfunktion) *209*
peek (Bibliotheksmakro) *209*
peekb (Bibliotheksfunktion) *209*
peekb (Bibliotheksmakro) *209*
PeekHead
TMDoubleListImp-Elementfunktion *430*
TMIDoubleListImp-Elementfunktion *435*
TMIListImp-Elementfunktion *449*
TMListImp-Elementfunktion *446*

PeekLeft
TMDequeAsDoubleList-Element-funktion *419*
TMDequeAsVector-Elementfunktion *413*
TMIDequeAsDoubleList-Element-funktion *421*
TMIDequeAsVector-Elementfunktion *417*
PeekRight
TMDequeAsDoubleList-Element-funktion *419*
TMIDequeAsDoubleList-Element-funktion *422*
TMIDequeAsVector-Elementfunktion *417*
PeekTail
TMDoubleListImp-Elementfunktion *431*
TMIDoubleListImp-Elementfunktion *435*
Peripheriegeräte
Liste ausgeben *57*
siehe Hardware *167*
perror (Bibliotheksfunktion) *209, 328*
Pfade
_searchenv *247*
searchpath *248*
Pfadnamen
erzeugen *123, 187*
relative konvertieren *138*
suchen *142*
zerlegen *124, 270*
Pipes
erzeugen *211*
PointerTypes
pstream-Datentyp *372*
poke (Bibliotheksfunktion) *212*
poke (Bibliotheksmakros) *212*
pokeb (Bibliotheksfunktion) *212*
pokeb (Bibliotheksmakro) *213*
polar
complex-Elementfunktion *494*
poly (Bibliotheksfunktion) *213*
polyl (Bibliotheksfunktion) *213*
Polynome *213*
Pop
TMIStackAsVector-Elementfunktion *467*
TMStackAsVector-Elementfunktion *464*
Ports
siehe Schnittstellen *161, 162*
Position
TFile-Elementfunktion *518*
Positionierung in einer Datei *232*
pow
complex-Elementfunktion *494*
pow (Bibliotheksfunktion) *214*
pow10 (Bibliotheksfunktion) *215*
pow10l (Bibliotheksfunktion) *215*
powl (Bibliotheksfunktion) *214*
pptr
streambuf-Elementfunktion *358*
precision
ios-Elementfunktion *347*
PRECONDITION (Diagnose-Makro) *498*
PRECONDITIONX (Diagnose-Makro) *500*
prepend
string-Elementfunktion *524*
Prev
TMDequeAsVector-Elementfunktion *414*
TMDoubleListElement-Datenelement *429*
Previous
TDate-Elementfunktion *514*
PrintDate
TTime-Elementfunktion *543*
printf (Bibliotheksfunktion) *216*
process.h (Header-Datei) *27*
Programme
Aufräumarbeiten *65*
beenden *107*
Jokerzeichen beim Aufruf *39*
laden und starten *104*
pausieren *264*
Prozeß-ID *149*
speicherresidente *66*
Start *104*
Programmsegment-Präfix *149*
Programmsegmentpräfix (PSP) *333*
des aktuellen Programms *332*
Prozeß-ID *149*
Prozeßroutinen
_c_exit *65*
_cexit *65*
_exit *107*
abort *45*
execl *104*
execle *104*
execlp *104*
execlpe *104*
execv *104*
execve *104*
execvp *104*
execvpe *104*

exit *107*
getpid *149*
spawnl *266*
spawnle *266*
spawnlp *266*
spawnlpe *266*
spawnv *266*
spawnve *266*
spawnvp *266*
spawnvpe *266*
Prozess-Routinen
system *296*
Prozesse
Eltern- *104, 267*
Funktionen zur Steuerung *35*
Kind- *104, 267*
Pseudo-Zufallszahlen *228*
PSP (Programmsegment-Präfix) *149*
psp_-Variable *149*
pstream (Klasse) *372*
Puffer
einem Stream zuordnen *258*
leeren *121*
Streams schließen *112*
Streams und *249*
Tastatur *310*
Pufferung von Ein-/Ausgaben *259*
Push
TMIStackAsVector-Elementfunktion *467*
TMStackAsVector-Elementfunktion *464*
put
ostream-Elementfunktion *354*
TMIQueueAsDoubleList-Elementfunktion *458*
TMIQueueAsVector-Elementfunktion *455*
TMQueueAsDoubleList-Elementfunktion *457*
TMQueueAsVector-Elementfunktion *453*
put_at
string-Elementfunktion *525*
TSubString -Elementfunktion *531*
putback
istream-Elementfunktion *351*
putc (Bibliotheksfunktion) *222*
putc (Bibliotheksmakro) *222*
putch (Bibliotheksfunktion) *222*
putchar (Bibliotheksmakro) *223*
putenv (Bibliotheksfunktion) *223*
PutLeft
TMDequeAsDoubleList-Elementfunktion *419*
TMDequeAsVector-Elementfunktion *413*
TMIDequeAsDoubleList-Elementfunktion *422*
TMIDequeAsVector-Elementfunktion *417*
PutRight
TMDequeAsDoubleList-Elementfunktion *419*
TMDequeAsVector-Elementfunktion *413*
TMIDequeAsDoubleList-Elementfunktion *422*
TMIDequeAsVector-Elementfunktion *417*
puts (Bibliotheksfunktion) *224*
puttext (Bibliotheksfunktion) *224*
putw (Bibliotheksfunktion) *225*

Q

qsort (Bibliotheksfunktion) *226*
Quadratwurzel *271*
QuickSort-Algorithmus *226*
Quotient *85*
zweier Integer *176*

R

rad (Winkeleinheit) *47, 50*
raise
xalloc-Elementfunktion *508*
xmsg-Elementfunktion *509*
raise (Bibliotheksfunktion) *227*
RAM
Größe *58, 59, 60*
rand (Bibliotheksfunktion) *227*
random (Bibliotheksmakro) *228*
randomize (Bibliotheksmakro) *228*
rdbuf
constream-Elementfunktion *339*
fpbase-Elementfunktion *364*
fstream-Elementfunktion *343*
fstreambase-Elementfunktion *344*
ifpstream-Elementfunktion *365*
ifstream-Elementfunktion *345*
ios-Elementfunktion *348*
ofpstream-Elementfunktion *369*

ofstream-Elementfunktion *353*
pstream-Elementfunktion *373*
strstreambase-Elementfunktion *359*
rdstate
ios-Elementfunktion *348*
pstream-Elementfunktion *373*
read
istream-Elementfunktion *351*
TFile-Elementfunktion *518*
TStreamer-Elementfunktion *377*
read (Bibliotheksfunktion) *229*
read_file
string-Elementfunktion *525*
read_line
string-Elementfunktion *525*
read_string
string-Elementfunktion *525*
read_to_delim
string-Elementfunktion *525*
read_token
string-Elementfunktion *525*
readByte
ipstream-Elementfunktion *366*
readData
ipstream-Elementfunktion *367*
readdir (Bibliotheksfunktion) *230*
readPrefix
ipstream-Elementfunktion *367*
readString
ipstream-Elementfunktion *366*
readSuffix
ipstream-Elementfunktion *367*
readVersion
ipstream-Elementfunktion *368*
readWord
ipstream-Elementfunktion *366*, *367*
readWord16
ipstream-Elementfunktion *367*
real
bcd-Elementfunktion *492*
realloc (Bibliotheksfunktion) *231*
Reallocate
TMArrayAsVector-Elementfunktion *387*
TMIArrayAsVector-Elementfunktion *392*
ref.h (Header-Datei) *27*
RefDate
TTime-Datenelement *541*
regexp.h (Header-Datei) *27*
Register
Segmentregister lesen *249*
registerObject
ipstream-Elementfunktion *367*
opstream-Elementfunktion *370*
Registervariablen
Task-Status und *252*
registerVB
opstream-Elementfunktion *370*
REGPACK-Struktur *167*
Remove
string-Elementfunktion *525*
TFile-Elementfunktion *518*
remove (Bibliotheksmakro) *231*
RemoveEntry
TMArrayAsVector-Elementfunktion *387*
TMIArrayAsVector-Elementfunktion *392*
Rename
TFile-Elementfunktion *518*
rename (Bibliotheksfunktion) *232*
replace
string-Elementfunktion *526*
requested
xalloc-Elementfunktion *508*
reserve
string-Elementfunktion *526*
Resize
string-Elementfunktion *526*
TMIVectorImp-Elementfunktion *480*
TMVectorImp-Elementfunktion *472*
resize_increment
string-Elementfunktion *526*
Rest
siehe Divisionsrest *122*
Restart
TBinarySearchTreeIteratorImp-Elementfunktion *409*
TIBinarySearchTreeIteratorImp-Elementfunktion *411*
TMArrayAsVectorIterator-Elementfunktion *388*
TMDequeAsVectorIterator-Elementfunktion *414*
TMDictionaryAsHashTableIterator-Elementfunktion *424*
TMDoubleListIteratorImp-Elementfunktion *431*
TMHashTableIteratorImp-Elementfunktion *440*

TMIArrayAsVectorIterator-Element-funktion *393*
TMIDictionaryAsHashTableIterator-Elementfunktion *427*
TMIDoubleListIteratorImp-Element-funktion *436*
TMIHashTableIteratorImp-Element-funktion *442*
TMIListIteratorImp-Elementfunktion *450*
TMIVectorIteratorImp-Elementfunktion *481*
TMListIteratorImp-Elementfunktion *447*
TMVectorIteratorImp-Elementfunktion *473*
Resume
TThread-Elementfunktion *539*
rewind (Bibliotheksfunktion) *232*
rewinddir (Bibliotheksfunktion) *233*
rfind
string-Elementfunktion *525*
Right
TMDequeAsVector-Datenelement *413*
rmdir (Bibliotheksfunktion) *233*
rmtmp (Bibliotheksfunktion) *234*
Rotation
long Integer *185*
unsigned char *80*
Rotation, bitweise *185*, *234*
Rundung *64*, *121*
von BCD-Zahlen *490*
Rundungsrichung
in Gleitkommaoperationen *73*

S

S_IFCHR, S_IFREG (Konstanten) *136*
S_IREAD *309*
S_IREAD, S_IWRITE (Konstanten) *68*, *136*
S_IWRITE *309*
sbumpc
streambuf-Elementfunktion *357*
scanf (Bibliotheksfunktion) *240*
Schließen
Datei *86*
Dateien *71*, *235*
Verzeichnis-Stream *71*
Schnittstelle
Überprüfung *58*
Überprüfung auf Vorhandensein *57*
Schnittstellen
Ausgabe zu *207*
Ein-/Ausgabe zu *161*, *162*
search.h (Header-Datei) *27*
searchpath (Bibliotheksfunktion) *248*
Second
TTime-Elementfunktion *543*
Seconds
TTime-Elementfunktion *543*
Seek
TFile-Elementfunktion *519*
SEEK_CUR, SEEK_END und SEEK_SET (Konstanten) *132*
seekg
ipstream-Elementfunktion *367*
istream-Elementfunktion *351*
seekoff
filebuf-Elementfunktion *341*
streambuf-Elementfunktion *357*
strstreambuf-Elementfunktion *360*
seekp
opstream-Elementfunktion *370*
ostream-Elementfunktion *354*
seekpos
streambuf-Elementfunktion *357*
SeekToBegin
TFile-Elementfunktion *519*
SeekToEnd
TFile-Elementfunktion *519*
Segment-Präfix *149*
Segmente
far-Zegier *198*
far-Zeiger *126*
Register lesen *249*
segread (Bibliotheksfunktion) *249*
Seicher
initialisieren *257*
set_case_sensitive
string-Elementfunktion *526*
set_new_handler (Funktion) *503*
set_paranoid_check
string-Elementfunktion *526*
set_terminate (Funktion) *505*
set_unexpected (Funktion) *505*
setb
streambuf-Elementfunktion *358*
setbuf
filebuf-Elementfunktion *342*
fpbase-Elementfunktion *364*
fstreambase-Elementfunktion *344*
streambuf-Elementfunktion *357*

strstreambuf-Elementfunktion *360*
setbuf (Bibliotheksfunktion) *249*
setcbrk (Bibliotheksfunktion) *250*
setcursortype
conbuf-Elementfunktion *338*
SetData
TMArrayAsVector-Elementfunktion *387*
TMIArrayAsVector-Elementfunktion *392*
setdate (Bibliotheksfunktion) *91*
setdisk (Bibliotheksfunktion) *145*
setdta (Bibliotheksfunktion) *251*
setf
ios-Elementfunktion *348*
setftime (Bibliotheksfunktion) *147*
setg
streambuf-Elementfunktion *358*
setjmp (Bibliotheksfunktion) *252*
setjmp.h (Header-Datei) *27*
setlocale (Bibliotheksfunktion) *253*
setmem (Bibliotheksfunktion) *257*
setmode (Bibliotheksfunktion) *257*
setp
streambuf-Elementfunktion *358*
SetPrintOption
TDate-Elementfunktion *514*
SetPriority
TThread-Elementfunktion *539*
setstate
ios-Elementfunktion *349*
pstream-Elementfunktion *374*
SetStatus
TFile-Elementfunktion *519*
settime (Bibliotheksfunktion) *152*
setvbuf (Bibliotheksfunktion) *258*
setvect (Bibliotheksfunktion) *152*
setverify (Bibliotheksfunktion) *259*
sgetc
streambuf-Elementfunktion *357*
sgetn
streambuf-Elementfunktion *357*
share.h (Header-Datei) *27*
SHARED-Modus *96, 133, 180, 238, 265*
ShouldTerminate
TThread-Elementfunktion *540*
sig_atomic_t (Datentyp) *261*
signal (Bibliotheksfunktion) *260*
Signal-Handler *227, 260*
signal.h (Header-Datei) *27*
sin
complex-Elementfunktion *495*
sin (Bibliotheksfunktion) *263*
sinh
complex-Elementfunktion *495*
sinh (Bibliotheksfunktion) *264*
sinhl (Bibliotheksfunktion) *264*
sinl (Bibliotheksfunktion) *263*

Sinus *263*
hyperbolicus *264*
skip_whitespace
string-Elementfunktion *526*
sleep (Bibliotheksfunktion) *264*
snextc
streambuf-Elementfunktion *357*
Software-Interrupt *163, 164*
Software-Interrupts *166*
erzeugen *139*
Software-Signale
senden *227*
signal *260*
Sommerzeit *327*
sopen (Bibliotheksmakro) *265*
Sortieren
Array-Elemente *226*
QuickSort *226*
Sortieren von Array-Elementen *226*
spawnl (Bibliotheksfunktion) *266*
spawnle (Bibliotheksfunktion) *266*
spawnlp (Bibliotheksfunktion) *266*
spawnlpe (Bibliotheksfunktion) *266*
spawnv (Bibliotheksfunktion) *266*
spawnve (Bibliotheksfunktion) *266*
spawnvp (Bibliotheksfunktion) *266*
spawnvpe (Bibliotheksfunktion) *266*
Speicher
Block kopieren *194*
Blöcke kopieren *195, 196*
Daten kopieren *200*
Daten verlagern *201*
DMA *57, 58, 59*
DTA *145*
freie Blöcke füllen *158, 159*
freie Blöcke überprüfen *156*
freigeben *129*
freigeben auf FAR-Heap *110*
Größe *60, 273*
Header-Datei *26*
Heap-Block *202*

initialisieren *197*
reservieren auf FAR-Heap *109, 111*
Reservierung *64*
Speichermodelle und *64*
unbenutzte Blöcke freigeben *158*
Speicherblöcke
Größe ändern *231*
Größe anpassen *111*
initialisieren *257*
suchen *195*
Speichermodelle *25*
DOS-Systemaufrufe und *53, 54*
Heap und *64*
Speicherreservierung und *111*
Speicherreservierung *34*
Speicherreservierung
Fehler *331*
Fehler bei *503, 508*
Speicherreservierung, dynamische *273*
speicherresidente Programme *66*
Speicherverwaltung
Funktionen zur *33, 34*
Sperren
entfernen *311*
sprintf (Bibliotheksfunktion) *271*
Sprünge
longjmp *183*
setjmp *252*
Sprungfunktionen *34*
sputbackc
streambuf-Elementfunktion *357*
sputc
streambuf-Elementfunktion *357*
sputn
streambuf-Elementfunktion *357*
sqrt
complex-Elementfunktion *495*
sqrt (Bibliotheksfunktion) *271*
sqrtl (Bibliotheksfunktion) *271*
SqueezeEntry
TMArrayAsVector-Elementfunktion *387*
TMIArrayAsVector-Elementfunktion *392*
srand (Bibliotheksfunktion) *272*
sscanf (Bibliotheksfunktion) *272*
Stack *64*
Größe *273*
Speicher reservieren auf *47*
Zeiger *252*
stackavail (Bibliotheksfunktion) *273*
Standard-Routinen
ltoa *187*
malloc *188*
putenv *223*
qsort *226*
rand *227*
swab *295*
system *296*
ultoa *308*
Standardroutinen
_atold *51*
_c_exit *65*
_cexit *65*
_exit *107*
abort *45*
abs *45*
atexit *51*
atof *51*
atoi *52*
atol *53*
bsearch *62*
calloc *64*
ecvt *101*
exit *107*
fcvt *113*
free *129*
gcvt *139*
getenv *146*
itoa *174*
labs *175*
lfind *176*
lsearch *185*
realloc *231*
srand *272*
strtod *290*
strtol *293*
start
TSubString -Elementfunktion *532*
TThread-Elementfunktion *539*
stat (Bibliotheksfunktion) *135*
stat-Struktur *135*
state
pstream-Elementfunktion *374*
Status-WORD *273*
Statuswort
löschen *69*
stdarg.h (Header-Datei) *27*
stddef.h (Header-Datei) *27*
stdin

formatierte Eingabe von *316*
lesen und formatieren aus *240*
Puffer und *249*
String lesen aus *150*
Zeichen lesen aus *116, 141*
stdio.h (Header-Datei) *27*
stdiostr.h (Header-Datei) *27*
stdlib.h (Header-Datei) *27*
stdout
formatierte Ausgabe zu *216, 315*
Puffer und *249*
Strings schreiben in *224*
Zeichen schreiben in *223*
Zeichen schreiben zu *128*
Steuerwort für Gleitkommaoperationen *73*
stime (Bibliotheksfunktion) *274*
stossc
streambuf-Elementfunktion *357*
stpcpy (Bibliotheksfunktion) *274*
str
ostrstream-Elementfunktion *356*
strstream-Elementfunktion *361*
strstreambuf-Elementfunktion *360*
strcat (Bibliotheksfunktion) *275*
strchr (Bibliotheksfunktion) *275*
strcmp (Bibliotheksfunktion) *276*

strcmpi (Bibliotheksfunktion) *276*
strcmpi (Bibliotheksmakro) *276*
strcoll (Bibliotheksfunktion) *277*
strcpy (Bibliotheksfunktion) *277*
strcspn (Bibliotheksfunktion) *278*
strdup (Bibliotheksfunktion) *278*
stream
clearerr *70*
Fehler *115*
feof *114*
ferror *115*
fgetc *116*
ftell *137*
fwrite *139*
getchar *141*
rewind *232*
StreamableName
TStreamer-Elementfunktion *377*
streambuf (Klasse) *356*
Streams
Ausgabepuffer in Datei schreiben *115*
Datei-Handle zuordnen *113*
Dateiende-Flag zurücksetzen *70*
Dateizeiger *132, 133*
Ein-/Ausgabe *114, 125, 130, 133*
Ende überprüfen *114*
Fehler-Flag zurücksetzen *70*
formatierte Ausgabe in *127*
formatierte Ausgabe zu *216, 314, 315*
formatierte Eingabe aus *131*
formatierte Eingabe von *314, 316*
Handle *118*
Integer lesen aus *154*
Integer schreiben in *225*
lesen aus *116, 129, 131, 150*
lesen aus Streams *117*
lesen und formatieren aus *240*
Makros für Streams *378*
öffnen *125, 130, 133*
Puffer leeren *121*
Puffer zuordnen *258*
schließen *112, 130*
schreiben in *139*
Strings ausgeben in *128*
Strings schreiben in *224*
ungepufferte *249*
Zeichen ausgeben zu *128*
Zeichen lesen aus *140, 309*
Zeichen schreiben in *222*
Zeiger initialisieren *232*
strerror (Bibliotheksfunktion) *280*
strftime (Bibliotheksfunktion) *280*
Strg-Untbr *250*
stricmp (Bibliotheksfunktion) *282*
String
Datum konvertieren in *278*
Klein- zu Großbuchstaben *294*
konvertieren in double *290*
konvertieren zu long double *290*
Länge eines Teilstrings *278*
mit Fehlermeldungen *279, 280*
setmem *257*
stpcpy *274*
vergleichen *276*
verketten *275*
zu long konvertieren *292*
zu unsigned long konvertieren *293*
String (Klasse) *519*
string.h (Header-Datei) *27*
Strings
absuchen *291*

anzeigen 77
aus Environment-Tabelle lesen 146
Ausgabe in einen Stream 128
durchsuchen 287
formatierte Ausgabe in 271, 317
formatierte Eingabe aus 317
Gleitkommazahl umwandeln zu 139
Gleitkommazahlen konvertieren zu 113
in stdout schreiben 224
kopieren 274, 277, 278, 285
Länge 283
lesen aus stdin 150
lesen aus Streams 117
lesen und formatieren aus 272
lesen von Tastatur 66
mit Zeichen füllen 286
Segmente suchen in 289
Teilstring sortieren 294
Teilstring suchen 289
überschreiben 288
Uhrzeit konvertieren zu 289
umdrehen 288
unsigned long zu 308
vergleichen 195, 196, 276, 277, 282, 284, 285, 286
verketten 284
Zeichen suchen in 275
strip
string-Elementfunktion 526
strlen (Bibliotheksfunktion) 283
strlwr (Bibliotheksfunktion) 283
strncat (Bibliotheksfunktion) 284
strncmp (Bibliotheksfunktion) 284
strncmpi (Bibliotheksmakro) 285
strncpy (Bibliotheksfunktion) 285
strnicmp (Bibliotheksfunktion) 286
strnset (Bibliotheksfunktion) 286
strpbrk (Bibliotheksfunktion) 287
strrchr (Bibliotheksfunktion) 287
strrev (Bibliotheksfunktion) 288
strset (Bibliotheksfunktion) 288
strspn (Bibliotheksfunktion) 289
strstr (Bibliotheksfunktion) 289
strstrea.h (Header-Datei) 27
strstream (Klasse) 361
strstreambase (Klasse) 359
strstreambuf (Klasse) 359
strtod (Bibliotheksfunktion) 290
strtok (Bibliotheksfunktion) 291
strtol (Bibliotheksfunktion) 292, 293
strtoul (Bibliotheksfunktion) 293
struct complex (Datentyp) 63
struct COUNTRY (Datentyp) 75
struct dfree (Datentyp) 144
struct diskfree_t (Datentyp) 92
struct DOSERROR (Datentyp) 88
struct dostime_t (Datentyp) 95
struct exception (Datentyp) 190
struct fatinfo (Datentyp) 146, 147
struct ffblk (Datentyp) 118
struct find_t (Datentyp) 89
struct ftime (Datentyp) 148
struct REGPACK (Datentyp) 167
struct stat (Datentyp) 135
struct time (Datentyp) 152
struct timeb (Datentyp) 137
struct tm (Datentyp) 154, 180
struct utimbuf (Datentyp) 312
Struktur
lokale 177
strupr (Bibliotheksfunktion) 294
strxfrm (Bibliotheksfunktion) 294
substr
string-Elementfunktion 526
substring
string-Elementfunktion 527
Suche
in Speicherblöcken 195
lineare 176, 185
Suchen
in Strings 287
Suchroutinen
bsearch 62
lfind 176
lsearch 185
Suchwege
siehe auch Pfadnamen 123
Suspend
TThread-Elementfunktion 539
swab (Bibliotheksfunktion) 295
sync
filebuf-Elementfunktion 342
strstreambuf-Elementfunktion 360
sync_with_stdio, ios-Elementfunktion 348
Syntaxfehler
siehe Fehler 295
sys\locking.h (Header-Datei) 27
sys\stat.h (Header-Datei) 27

sys\timeb.h (Header-Datei) 27
sys\types.h (Header-Datei) 27
System
 Datum und Uhrzeit 94, 152, 303
 Datum und Uhrzeit setzen 274
 Fehlermeldungen 209, 328
 Puffer 112
system (Bibliotheksfunktion) 296

T

Tabellen
 absuchen 62
 durchsuchen 185
Tail
 TMDoubleListImp-Datenelement 431
 TMListImp-Datenelement 446
tan
 complex-Elementfunktion 495
tan (Bibliotheksfunktion) 296
Tangens 296
Tangens hyperbolicus 297
tanh
 complex-Elementfunktion 495
tanh (Bibliotheksfunktion) 297
tanhl (Bibliotheksfunktion) 297
tanl (Bibliotheksfunktion) 296
TArrayAsVector (Template) 388
TArrayAsVectorIterator (Template) 389
Task-Status 252
Task-Zustand 183
Tastatur
 Abfrage 59
 Ein-/Ausgabe 141, 175
 formatierte Eingabe von 80
 lesen von 66, 80
 Zeichen lesen von 142
Tastaturpuffer
 Zeichen lesen von 310
Tastendruck, Test auf 175
TBagAsVector (Template) 404
TBagAsVectorIterator (Template) 404
TBinarySearchTreeImp (Template) 407
TBinarySearchTreeIteratorImp (Template) 408
TCriticalSection (Klasse) 533
TCriticalSection::Lock (Klasse) 533
TCVectorImp (Template) 477
TCVectorIteratorImp (Template) 477
TDate (Klasse) 511
TDDAssociation (Template) 397
TDequeAsDoubleList (Template) 420
TDequeAsDoubleListIterator (Template) 420
TDequeAsVector (Template) 415
TDequeAsVectorIterator (Template) 415
TDIAssociation (Template) 399
TDictionary (Template) 428
TDictionaryAsHashTable (Template) 425
TDictionaryAsHashTableIterator
 (Template) 425
TDictionaryIterator (Template) 428
TDoubleListImp (Template) 432
TDoubleListIteratorImp (Template) 432
tell (Bibliotheksfunktion) 297
tellg
 ipstream-Elementfunktion 367
 istream-Elementfunktion 352
tellp
 opstream-Elementfunktion 370
 ostream-Elementfunktion 355
TEMP (Umgebungsvariable) 304
template
 Dateinamen 199
tempnam (Bibliotheksfunktion) 298
temporäre Dateien
 löschen 234
 Namensvergabe 298, 304
 öffnen 304
 siehe auch Dateien 301
Terminate
 TThread-Elementfunktion 539
terminate (Funktion) 506
TerminateAndWait
 TThread-Elementfunktion 539
Terminierungsroutinen 51
Text
 Attribute setzen 299, 300, 301
 auf Bildschirm ausgeben 225
 Ausschnitt kopieren 201
 in Speicher kopieren 150
 Intensität 184, 202
 Zeichenfarbe 299
 Zeichenfarbe setzen 301
 Zeichenintensität 160
 Zeile löschen 84
text_info (Datentyp) 151
textattr
 conbuf-Elementfunktion 338

textattr (Bibliotheksfunktion) *299*
Textattribute
 setzen *160, 184, 202*
Textausgabe
 BIOS vs. direkter Zugriff *327*
 Parameter *151*
textbackground (Bibliotheksfunktion) *300*
textbackgroundcolor
 conbuf-Elementfunktion *338*
Textbildschirmanwendung
 Funktionen für *29*
textcolor
 conbuf-Elementfunktion *338*
textcolor (Bibliotheksfunktion) *301*
Textdatei
 lesen *229*
Textdateien
 _rtl_read und *239*
 creat und *77*
 creattemp und *79*
 fdopen *114*
 fopen und *125*
 freopen und *130*
 modus setzen *257*
 siehe auch Dateien, Dateibearbeitung *77*
Textfenster
 Cursorposition *155*
 Cursorposition im *320, 321*
 Leerzeile einfügen *163*
 setzen *321*
 Zeile löschen *72, 84*
textmode
 conbuf-Elementfunktion *338*
 constream-Elementfunktion *339*
textmode (Bibliotheksfunktion) *302*
Textmodus *204, 225, 240, 265, 321, 322*
 aktueller *151*
 Dateien öffnen im *331*
 Koordinaten *150*
 setzen *302*
Datei-Flags *516*
TFileStatus (Struktur) *516*
THashTableImp (Template) *440*
THashTableIteratorImp (Template) *441*
Threads
 Erzeugen von *41*
 Programme mit mehreren *41*
 starten *55*
TIArrayAsVector (Template) *394*
TIArrayAsVectorIterator (Template) *394*
TIBagAsVector (Template) *407*
TIBagAsVectorIterator (Template) *407*
TIBinarySearchTreeImp (Template) *409*
TIBinarySearchTreeIteratorImp (Template) *410*
TICVectorImp (Template) *484*
TICVectorIteratorImp (Template) *484*
TIDAssociation (Template) *400*
TIDequeAsDoubleList (Template) *422*
TIDequeAsDoubleListIterator (Template) *422*
TIDequeAsVector (Template) *417*
TIDequeAsVectorIterator (Template) *418*
TIDictionaryAsHashTable (Template) *427*
TIDictionaryAsHashTableIterator (Template) *428*
TIDoubleListImp (Template) *436*
TIDoubleListIteratorImp (Template) *437*
tie
 ios-Elementfunktion *348*
TIHashTableImp (Template) *443*
TIHashTableIteratorImp (Template) *443*
TIIAssociation (Template) *401*
TIListImp (Template) *450*
TIListIteratorImp (Template) *450*
time (Bibliotheksfunktion) *303*
time.h (Header-Datei) *27*
timezone (globale Variable) *334*
TIQueueAsDoubleList (Template) *459*
TIQueueAsDoubleListIterator (Template) *459*
TIQueueAsVector (Template) *455*
TIQueueAsVectorIterator (Template) *456*
TISArrayAsVector (Template) *396*
TISArrayAsVectorIterator (Template) *396*
TISDoubleListImp (Template) *438*
TISDoubleListIteratorImp (Template) *438*
TISetAsVector (Template) *462*
TISetAsVectorIterator (Template) *462*
TISListImp (Template) *451*
TISListIteratorImp (Template) *452*
TIStackAsList (Template) *470*
TIStackAsListIterator (Template) *470*
TIStackAsVector (Template) *467*
TIStackAsVectorIterator (Template) *468*
TISVectorImp (Template) *485*
TISVectorIteratorImp (Template) *486*
TIVectorImp (Template) *482*
TIVectorIteratorImp (Template) *482*
TListImp (Template) *447*
TListIteratorImp (Template) *447*

TMArrayAsVector (Template) *383*
TMArrayAsVectorIterator (Template) *387*
TMBagAsVector (Template) *402*
TMBagAsVectorIterator (Template) *404*
TMCVectorImp (Template) *475*
TMCVectorIteratorImp (Template) *476*
TMDDAssociation (Template) *397*
TMDequeAsDoubleList (Template) *418*
TMDequeAsDoubleListIterator (Template) *419*
TMDequeAsVector (Template) *411*
TMDequeAsVectorIterator (Template) *414*
TMDIAssociation (Template) *398*
TMDictionaryAsHashTable (Template) *423*
TMDictionaryAsHashTableIterator
(Template) *424*
TMDoubleListElement (Template) *428*
TMDoubleListImp (Template) *429*
TMDoubleListIteratorImp (Template) *431*
TMHashTableImp (Template) *438*
TMHashTableIteratorImp (Template) *439*
TMIArrayAsVector (Template) *389*
TMIArrayAsVectorIterator (Template) *393*
TMIBagAsVector (Template) *405*
TMIBagAsVectorIterator (Template) *406*
TMICVectorImp (Template) *482*
TMICVectorIteratorImp (Template) *483*
TMIDAssociation (Template) *399*
TMIDequeAsDoubleList (Template) *420*
TMIDequeAsDoubleListIterator (Template) *422*
TMIDequeAsVector (Template) *415*
TMIDequeAsVectorIterator (Template) *417*
TMIDictionaryAsHashTable (Template) *425*
TMIDictionaryAsHashTableIterator
(Template) *426*
TMIDoubleListImp (Template) *434*
TMIDoubleListIteratorImp (Template) *436*
TMIHashTableImp (Template) *441*
TMIHashTableIteratorImp (Template) *442*
TMIIAssociation (Template) *401*
TMIListImp (Template) *448*
TMIListIteratorImp (Template) *450*
TMIQueueAsDoubleList (Template) *458*
TMIQueueAsDoubleListIterator
(Template) *459*
TMIQueueAsVector (Template) *454*
TMIQueueAsVectorIterator (Template) *455*
TMISArrayAsVector (Template) *396*
TMISDoubleListImp (Template) *437*
TMISDoubleListIteratorImp (Template) *437*
TMISetAsVector (Template) *461*
TMISetAsVectorIterator (Template) *462*
TMISListImp (Template) *451*
TMISListIteratorImp (Template) *451*
TMIStackAsList (Template) *469*
TMIStackAsListIterator (Template) *469*
TMIStackAsVector (Template) *466*
TMIStackAsVectorIterator (Template) *467*
TMISVectorImp (Template) *484*
TMISVectorIteratorImp (Template) *485*
TMIVectorImp (Template) *479*
TMIVectorIteratorImp (Template) *481*
TMListElement (Template) *444*
TMListImp (Template) *444*
TMListIteratorImp (Template) *446*
tmpfile (Bibliotheksfunktion) *304*
tmpnam (Bibliotheksfunktion) *304*
TMQueueAsDoubleList (Template) *456, 457*
TMQueueAsVector (Template) *452*
TMQueueAsVectorIterator (Template) *453*
TMSArrayAsVector (Template) *394*
TMSArrayAsVectorIterator (Template) *395*
TMSDoubleListImp (Template) *433*
TMSDoubleListIteratorImp (Template) *433*
TMSListImp (Template) *448*
TMSListIteratorImp (Template) *448*
TMStackAsList (Template) *468*
TMStackAsListIterator (Template) *468*
TMStackAsVector (Template) *463*
TMSVectorImp (Template) *478*
TMSVectorIteratorImp (Template) *478*
TMutex (Klasse) *534*
TMutex::Lock (Klasse) *535*
TMVectorImp (Template) *470*
TMVectorIteratorImp (Template) *473*
to_lower
string-Elementfunktion *527, 531*
TSubString -Elementfunktion *532*
to_upper
string-Elementfunktion *527, 531*
TSubString -Elementfunktion *532*
toascii (Bibliotheksmakro) *305*
tolower (Bibliotheksfunktion) *306*
Top
TMCVectorImp-Elementfunktion *476*
TMIStackAsVector-Elementfunktion *467*
TMIVectorImp-Elementfunktion *480*
TMStackAsVector-Elementfunktion *464*
TMVectorImp-Elementfunktion *472*

toupper (Bibliotheksfunktion) *307*
TQueue (Template) *459*
TQueueAsDoubleList (Template) *457*
TQueueAsDoubleListIterator (Template) *457*
TQueueAsVector (Template) *453*
TQueueAsVectorIterator (Template) *454*
TQueueIterator (Template) *460*
TRACE (Diagnose-Makro) *498*
TRACEX (Diagnose-Makro) *500*
trigonometrische Funktionen *46, 48, 49, 50, 263, 296*
 Cosinus *74*
 hyperbolsicher Cosinus *74*
 hypot *160*
 hypotl *160*
 Tangens *296, 297*
TSArrayAsVector (Template) *395*
TSArrayAsVectorIterator (Template) *395*
TSDoubleListImp (Template) *433*
TSDoubleListIteratorImp (Template) *434*
TSet (Template) *463*
TSetAsVector (Template) *461*
TSetAsVectorIterator (Template) *461*
TSetIterator (Template) *463*
TShouldDelete (Klasse) *486*
TSListImp (Template) *448*
TSListIteratorImp (Template) *448*
TSR-Programme *66*
TStack (Template) *470*
TStackAsList (Template) *469*
TStackAsListIterator (Template) *469*
TStackAsVector (Template) *465*
TStackAsVectorIterator (Template) *465*
TStackIterator (Template) *470*
TStreamableBase (Klasse) *374*
TStreamableClass (Klasse) *375*
TStreamer (Klasse) *376*
TSubString (Klasse) *531*
TSVectorImp (Template) *478*
TSVectorIteratorImp (Template) *479*
TSync (Klasse) *535*
TSync::Lock (Klasse) *537*
TThread (Klasse) *537*
TThread::TThreadError (Klasse) *540*
TTime (Klasse) *541*
TVectorImp (Template) *474*
TVectorIteratorImp (Template) *474*
Type_id
 TStreamableBase-Datentyp *375*
Type_info (Klasse) *506*
typeinfo.h (Header-Datei) *28*
Typinformationen *506*
tzname (globale Variable) *334*
tzset (Bibliotheksfunktion) *307*

U

Überschreiben
 Dateien *77*
Übersetzungsmodus
 festlegen *257*
Uhrzeit
 Differenz zweier Zeitangaben *84*
 formatieren für Ausgabe *280*
 konvertieren in String *289*
 siehe auch Datum und Uhrzeit *61*
 System- *81*
Uhrzeit und Datum
 _daylight (globale Variable) *327*
ultoa (Bibliotheksfunktion) *308*
umask (Bibliotheksfunktion) *309*
Umgebungsvariable
 COMSPEC *296*
 dem Environment hinzufügen *223*
 TEMP *304*
 Variable _environ *327*
Umwandlung
 Datum und Zeit in String *81*
 Datum/Uhrzeit in GMT *154*
 Datum/Zeit in Kalenderformat *199*
 DOS- in UNIX-Format *99*
 double in Ganzzahl und Bruchteil *200*
 Formatierungsanweisungen *216*
 Gleitkommazahl in String *139*
 Gleitkommazahl zu String *101, 113, 139*
 Groß- in Kleinbuchstaben *283, 305, 306*
 Integer zu String *174*
 Klein- zu Großbuchstaben *294, 306, 307*
 long-Wert zu String *187*
 siehe auch Konvertierungsroutinen *101*
 String in Integerzahl *52*
 String in long-Wert *53*
 String zu double *290*
 String zu long *292*
 String zu long double *290*
 String zu unsigned long *293*
 Strings in Gleitkommazahl *51*

Uhrzeit in String *289*
UNIX- zu DOS-Format *310*
unsigned long in String *308*
Zeichen in ASCII-Format *305*
unbuffered
streambuf-Elementfunktion *358, 359*
underflow
filebuf-Elementfunktion *342*
strstreambuf-Elementfunktion *360*
unendliche Werte
in Gleitkommaoperationen *73*
Unendlichkeiten
und atof *52*
unexpected (Funktion) *507*
ungepufferte Ein-/Ausgaben *259*
ungepufferte Streams *249, 258*
ungetc (Bibliotheksfunktion) *309*
ungetch (Bibliotheksfunktion) *310*
UNIX
Datum und Uhrzeit *99*
Datum und Uhrzeit konvertieren *310*
unixtodos (Bibliotheksfunktion) *310*
unlink (Bibliotheksfunktion) *311*
unlock (Bibliotheksfunktion) *311*
UnlockRange
TFile-Elementfunktion *519*
unsetf
ios-Elementfunktion *348*
Upperbound
TMArrayAsVector-Elementfunktion *386*
TMIArrayAsVector-Elementfunktion *392*
USA
Datumsformat *76*
utime (Bibliotheksfunktion) *312*
utime.h (Header-Datei) *28*

V

va_arg (Bibliotheksmakro) *313*
va_arg (Bibliotheksmakros
Makros *313*
va_end (Bibliotheksmakro) *313*
va_end (Bibliotheksmakros
Makros *313*
va_start (Bibliotheksmakro) *313*
va_start (Bibliotheksmakros
Makros *313*
valid_element
string-Elementfunktion *527*
valid_index
string-Elementfunktion *528*
Value
TMDDAssociation-Elementfunktion *397*
TMDIAssociation-Elementfunktion *398*
TMIDAssociation-Elementfunktion *400*
TMIIAssociation-Elementfunktion *401*
values.h (Header-Datei) *28*
varargs.h (Header-Datei) *28*
Variable Argumentlisten *36*
variable Parameterzahl *313*
Variablen
Umgebungs- *268*
Vergleich
zweier Werte *197*
Vergleichsroutine, selbstgeschriebene *62, 186*
Verify-Flag *153*
Versionsnummern
DOS *332, 334*
Vertauschen der Byte-Reihenfolge *295*
Verzeichnis
absuchen *89, 90*
aktuelle wechseln *67*
aktuelles *106, 142, 143*
aktuelles ermitteln *142*
durchsuchen *230*
erzeugen *198*
löschen *233*
Verzeichnis-Routinen
mkdir *198*
mktemp *199*
Verzeichnis-Stream
lesen aus *230*
schließen *71*
zurücksetzen *233*
Verzeichnis-Streams
öffnen *205*
Verzeichnispfad
durchsuchen *247, 248*
Verzeichnisroutinen
_dos_findfirst *89*
_dos_findnext *90*
_getdcwd *143*
_makepath *187*
_searchenv *247*
_searchstr *248*
_splitpath *270*
chdir *67*
findfirst *118*

findnext *120*
fnmerge *123*
fnsplit *124*
getcurdir *142*
getcwd *142*
getdisk *145*
rmdir *233*
rmtmp *234*
searchpath *248*
setdisk *145*
Verzeichnisse
absuchen *118, 120*
Funktionen zur Verwaltung *29*
vfprintf (Bibliotheksfunktion) *314*
vfscanf (Bibliotheksfunktion) *314*
vollständige Pufferung *259*
vprintf (Bibliotheksfunktion) *315*
vscanf (Bibliotheksfunktion) *316*
vsprintf (Bibliotheksfunktion) *317*
vsscanf (Bibliotheksfunktion) *317*

W

Währungssymbol *75*
wait (Bibliotheksfunktion) *318*
WaitForExit
TThread-Elementfunktion *539*
WARN (Diagnose-Makro) *498*
WARNX (Diagnose-Makro) *500*
wchar_t (Array) *319*
wcstombs (Bibliotheksfunktion) *319*
wctomb (Bibliotheksfunktion) *320*
WeekDay
TDate-Elementfunktion *514*
Werte
absolute *109*
konvertieren in ASCII-Format *305*
Literale *102*
Potenz berechnen *214*
Speicherbereich füllen mit *257*
vergleichen *191, 197*
wherex
conbuf-Elementfunktion *339*
wherex (Bibliotheksfunktion) *320*
wherey
conbuf-Elementfunktion *339*
wherey (Bibliotheksfunktion) *321*
why
xmsg-Elementfunktion *509*
width
ios-Elementfunktion *348*
WILDARGS.OBJ (für Jokerzeichen) *39*
window
conbuf-Elementfunktion *339*
constream-Elementfunktion *339*
window (Bibliotheksfunktion) *321*
Winkeleinheit rad *47, 50*
Wort
aus Schnittstelle lesen *162*
Ausgabe zu I/O-Adresse *207*
Dateiattribut- *79*
lesen aus Speicher *209*
Status- *273*
Steuer- *73*
wortweise Ausrichtung
Vertauschen von Bytes *295*
write
ostream-Elementfunktion *355*
TFile-Elementfunktion *519*
TStreamer-Elementfunktion *377*
write (Bibliotheksfunktion) *322*
writeByte
opstream-Elementfunktion *370*
writeBytes
opstream-Elementfunktion *370*
writeData
opstream-Elementfunktion *371*
writeObject
opstream-Elementfunktion *370*
writeObjectPtr
opstream-Elementfunktion *370*
writePrefix
opstream-Elementfunktion *371*
writeString
opstream-Elementfunktion *370*
writeSuffix
opstream-Elementfunktion *371*
writeWord
opstream-Elementfunktion *371*
writeWord16
opstream-Elementfunktion *371*
writeWord32
opstream-Elementfunktion *371*

X

xalloc

ios-Elementfunktion *349*
xalloc (Klasse) *508*
xmsg (Klasse) *508*

Y

Year
TDate-Elementfunktion *514*
YearTy
TDate-Datentyp *512*

Z

Zahlen
runden *121*
Rundung *64*
Umwandlung von Strings in *51*
Zufalls- *228*
Zeichen
Attribute *326*
Attribute setzen *299, 300, 301*
aus mehreren Bytes *192, 193, 319*
Ausgabe *128, 222, 223*
Farbe setzen *299*
in Strings überschreiben *288*
Klassifizierung *28, 153ff.*
konvertieren in ASCII-Format *305*
konvertieren zu Großbuchstaben *306, 307*
konvertieren zu Kleinbuchstaben *305, 306*
lesen aus stdin *116, 141*
lesen aus Stream *140*
lesen aus Streams *116*
lesen von Tastatur *66, 141, 142*
niedrige Intensität *184*
normale Intensität *202*
suchen in Speicherblöcken *195*
suchen in String *275*
suchen in Strings *287*
Zeilenvorschub *224*
Zeichenfarbe
für Textausgaben *301*
zeichenorientierte Geräte *250, 258*
Zeiger
auf Fehlermeldungen *279*
auf lokale Struktur *177*
Datei *96, 132, 133, 137, 238, 266, 297*
Datei- *116, 186, 229, 232, 239*
far *110, 111, 198*
far- *111*
Frame *252*
Zeile
einfügen *163*
löschen *72*
Zeilen
löschen *84*
Zeilenvorschub
"Übersetzung" *240, 322*
Zeilenvorschubzeichen *224*
zeilenweise Pufferung *259*
Zeit
siehe auch Datum und Uhrzeit *61*
Zeitangaben
Funktionen *36*
Zeitmessungen
clock *70*
Zeitzone
setzen *82*
Zeitzonen *137, 307*
tzname (glob. Variable) *334*
Zero
TMIVectorImp-Elementfunktion *480*
TMVectorImp-Elementfunktion *473*
ZeroBase
TMArrayAsVector-Elementfunktion *387*
TMIArrayAsVector-Elementfunktion *393*
Ziffern
prüfen auf *171*
Zufallszahlen *227, 228, 272*
Zugriff
lesen/schreiben *135*
Zugriffsberechtigung *234*
ändern *68*
Zugriffsmodus
Datei *93*
Zugriffsmodus (Dateien) *309*
Zuordnung
Unterdrückung bei scanf *245*
Zurückstellen von Zeichen *310*

Borland

Copyright © 1994, Borland International, Inc. Alle Rechte Vorbehalten. Alle Produktnamen von Borland sind eingetragene Warenzeichen der Borland International Inc.
Deutsche Ausgabe © 1994 Borland GmbH, Monzastraße 4c, D-63225 Langen, Telefon 06103/979-0, Fax 06103/979-290
Zweigniederlassung Schweiz, Solothurnstraße 31, CH-3303 Jegenstorf
Hauptsitz: 100 Borland Way, Scotts Valley, CA 95066-3249, +1 (408) 431-1000
Niederlassungen in: Australien, Belgien, Brasilien, Chile, Deutschland, Dänemark, Frankreich, Großbritannien, Hongkong, Italien, Japan, Kanada, Korea, Lateinamerika, Malaysia, Niederlande, Neuseeland, Schweden, Singapur, Spanien und Taiwan.
• Part # BCP1245GExxxxx